

IS 6733

Deep Learning on Cloud Platforms

Lecture 5

Neural Network Basics

Dr. Yuanxiong Guo
Department of Information Systems and Cyber Security



Acknowledgement

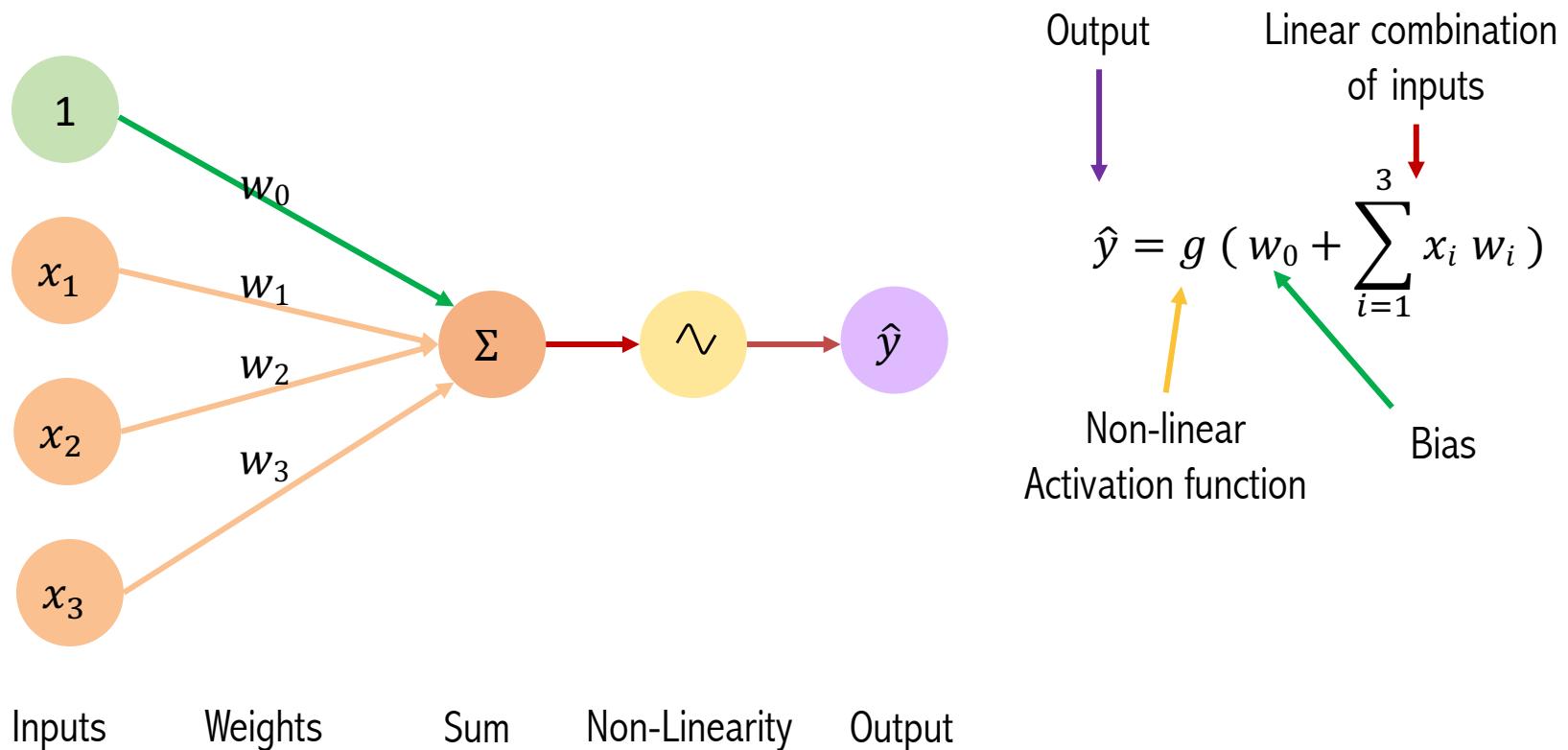
- Slides are adapted from the following open course:
 - Alexander Amini and Ava Soleimany,
MIT 6.S191: Introduction to Deep Learning
IntroToDeepLearning.com
- Thanks to Alexander and Ava for sharing the slides.

A first look at a neural network using Keras

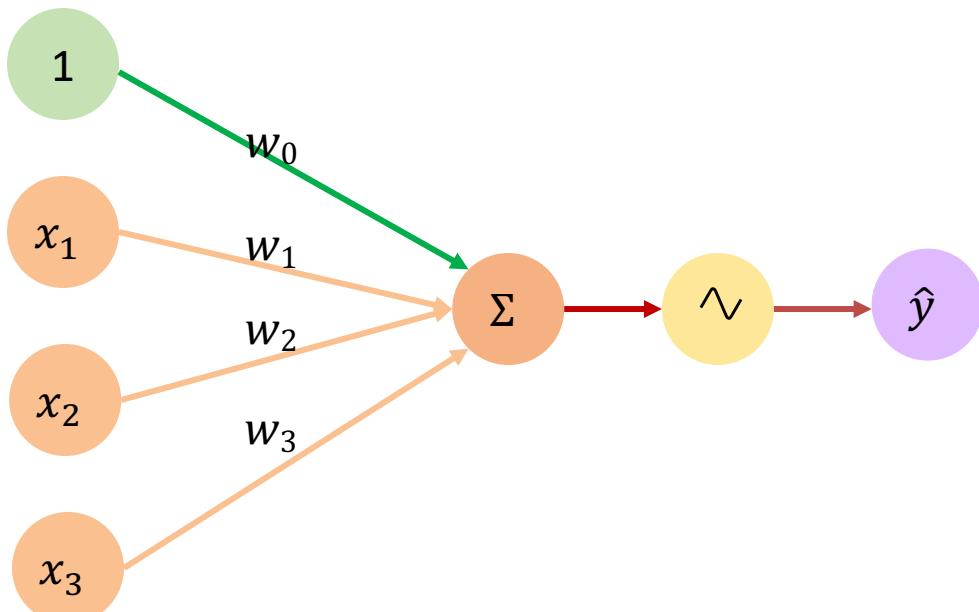
<https://colab.research.google.com/drive/13cof4XUULbUqO0s5h-cd17FskWjpyMkm>

The Perceptron: The structural building block of deep learning

The Perceptron: Forward Propagation



The Perceptron: Forward Propagation

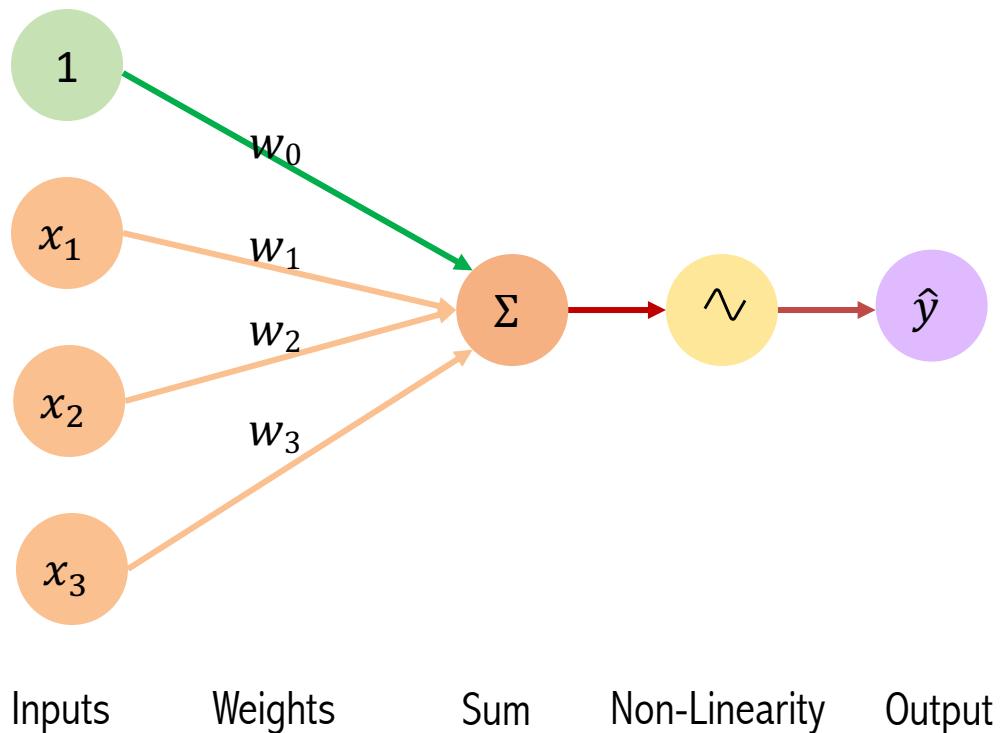


$$\hat{y} = g \left(w_0 + \sum_{i=1}^3 x_i w_i \right)$$

$$\hat{y} = g (w_0 + \mathbf{X}^T \mathbf{W})$$

where $\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ and $\mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$

The Perceptron: Forward Propagation

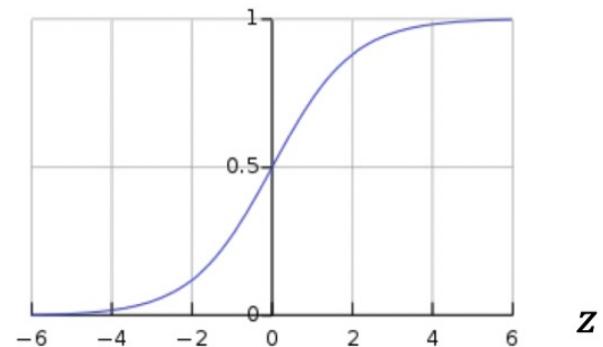


Activation Functions

$$\hat{y} = g(w_0 + X^T W)$$

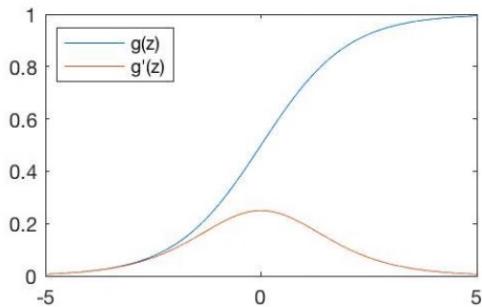
Example: sigmoid function

$$g(z) = \frac{1}{1 + e^{-x}}$$



Common Activation Functions

Sigmoid Function

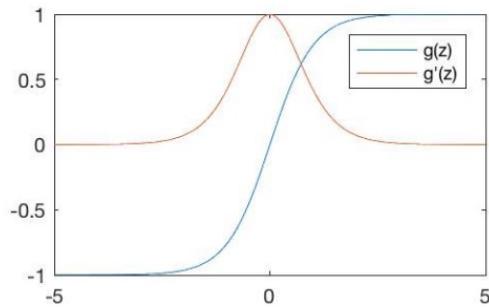


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

 `tf.nn.sigmoid(z)`

Hyperbolic Tangent

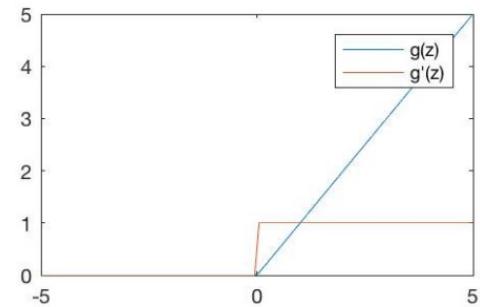


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

 `tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

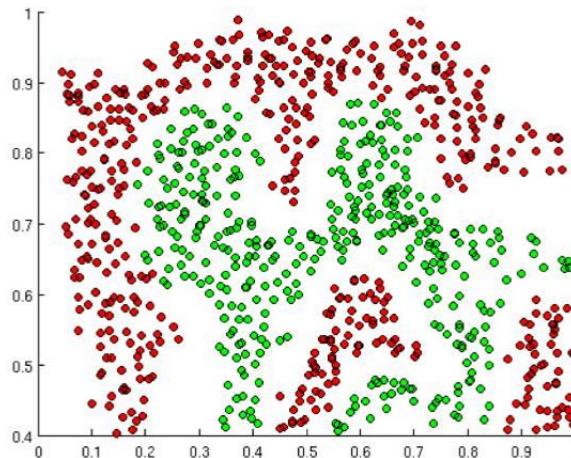
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

NOTE: All activation functions are non-linear

Importance of Activation Functions

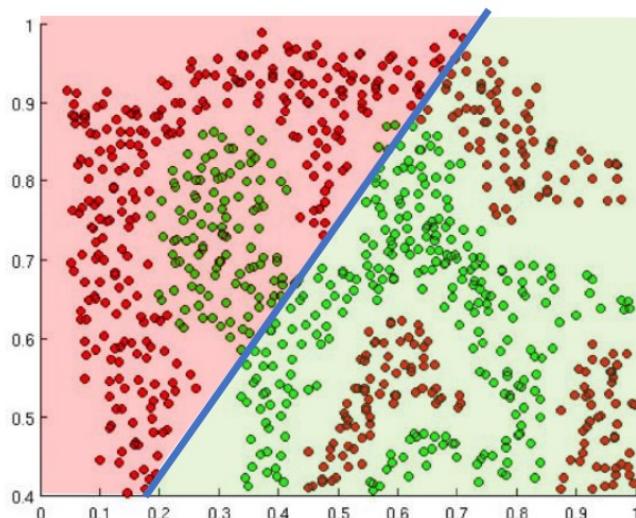
*The purpose of activation functions is to **introduce non-linearities** into the network*



What if we wanted to build a Neural Network to
distinguish green vs red points?

Importance of Activation Functions

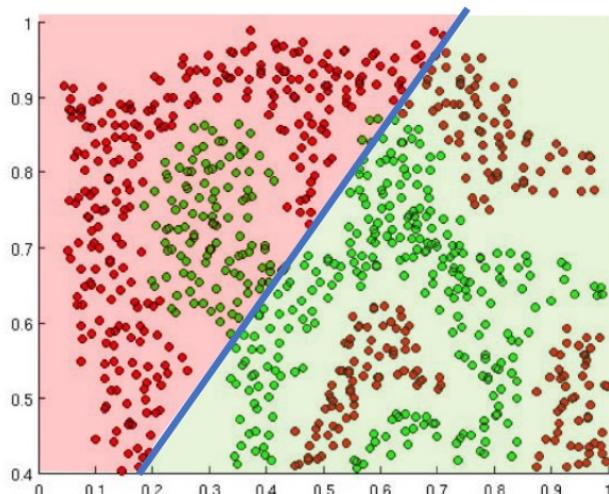
The purpose of activation functions is to **introduce non-linearities** into the network



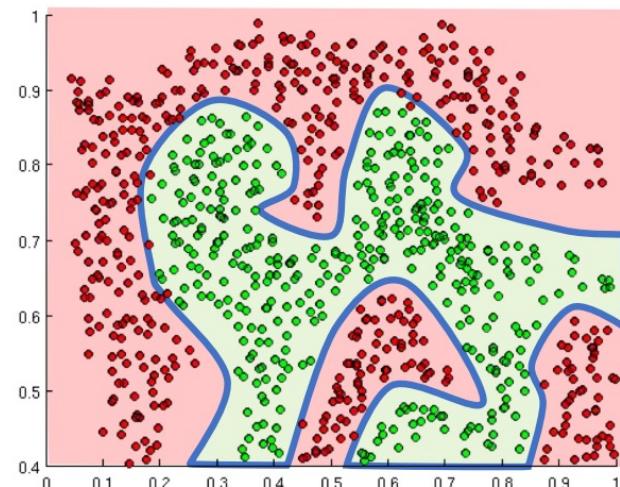
Linear Activation functions produce linear decisions no matter the network size

Importance of Activation Functions

The purpose of activation functions is to **introduce non-linearities** into the network

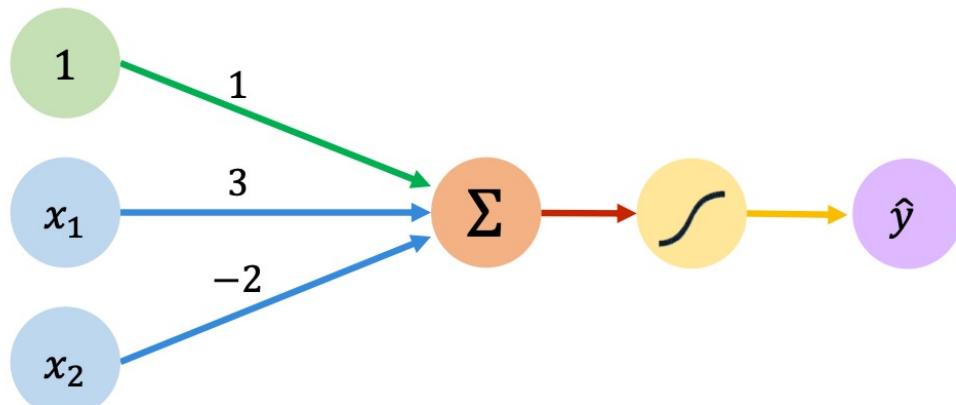


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example

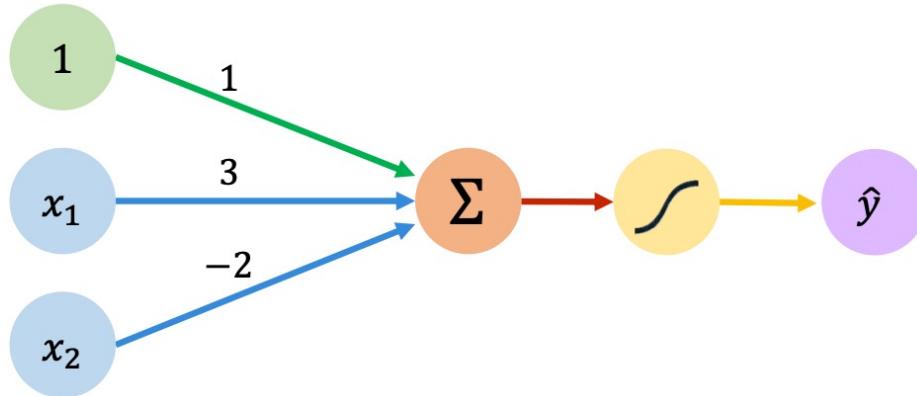


We have: $w_0 = 1$ and $\mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

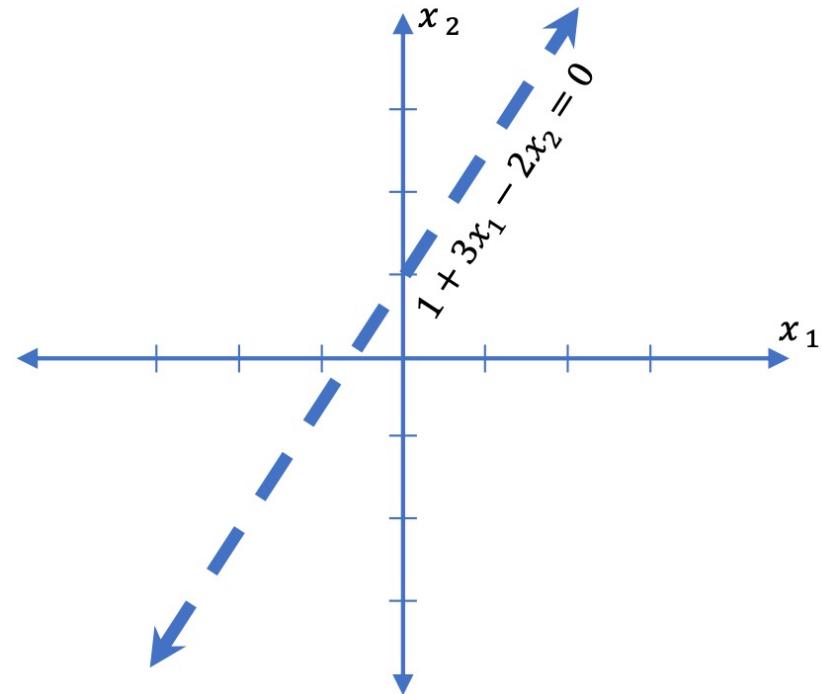
$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{w}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g\left(1 + 3x_1 - 2x_2\right)\end{aligned}$$

This is just a line in 2D!

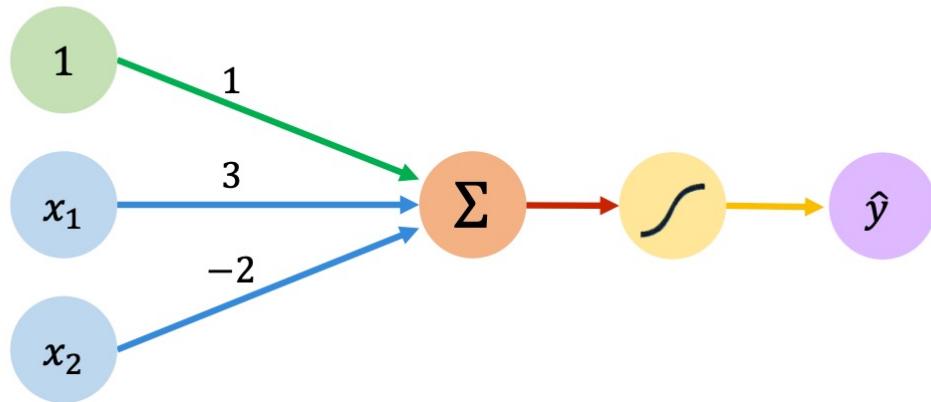
The Perceptron: Example



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



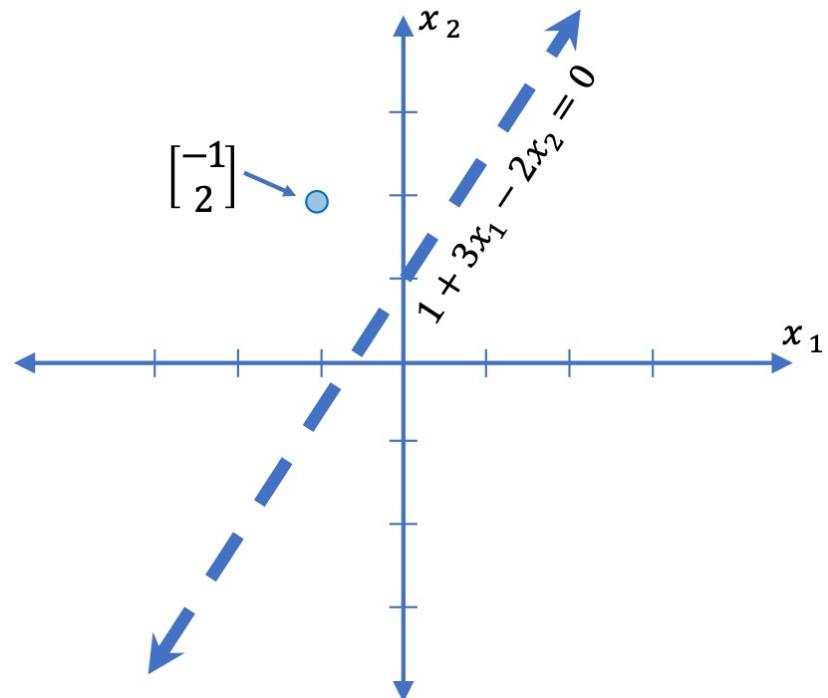
The Perceptron: Example



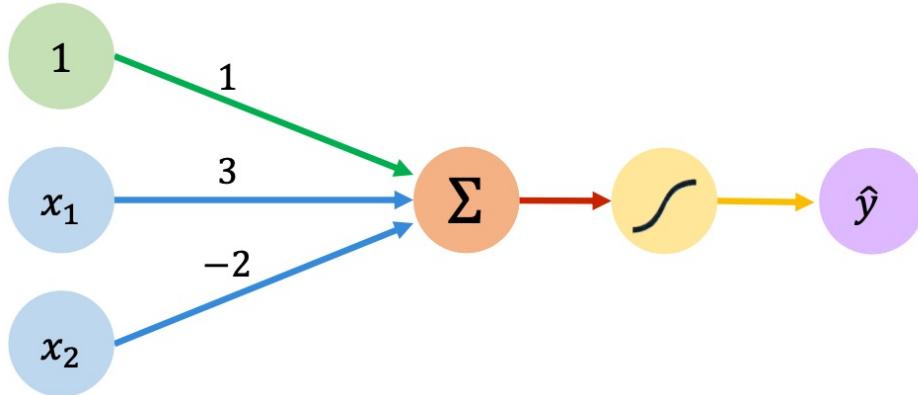
Assume we have input: $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$

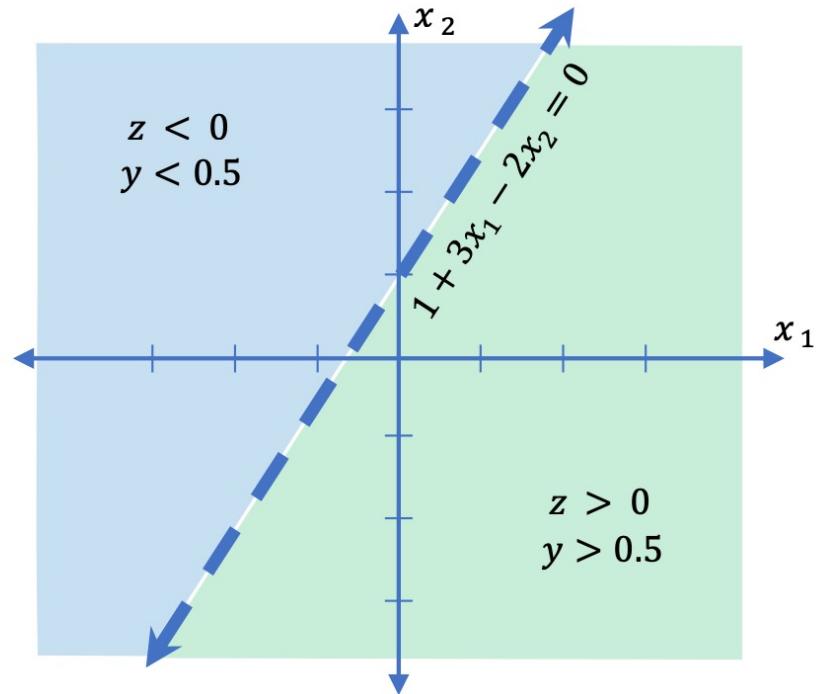
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron: Example

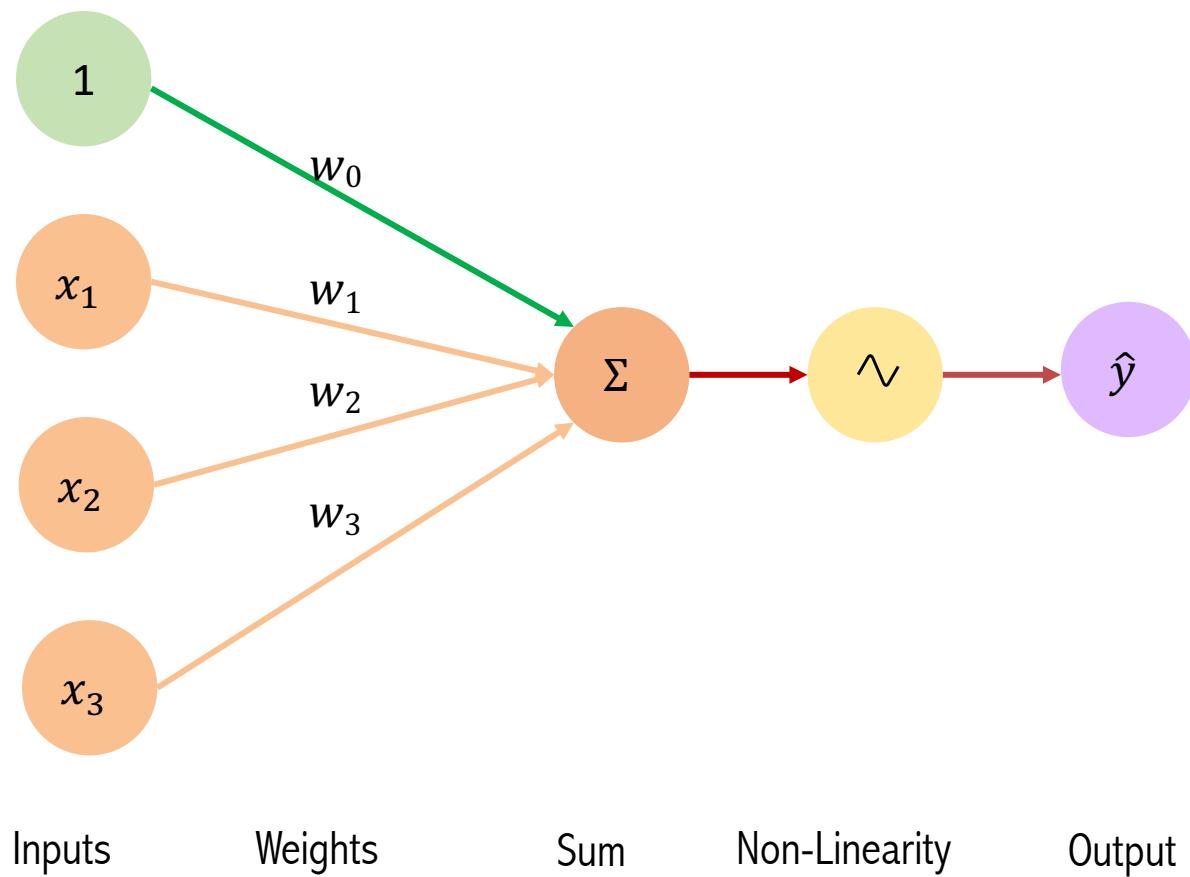


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

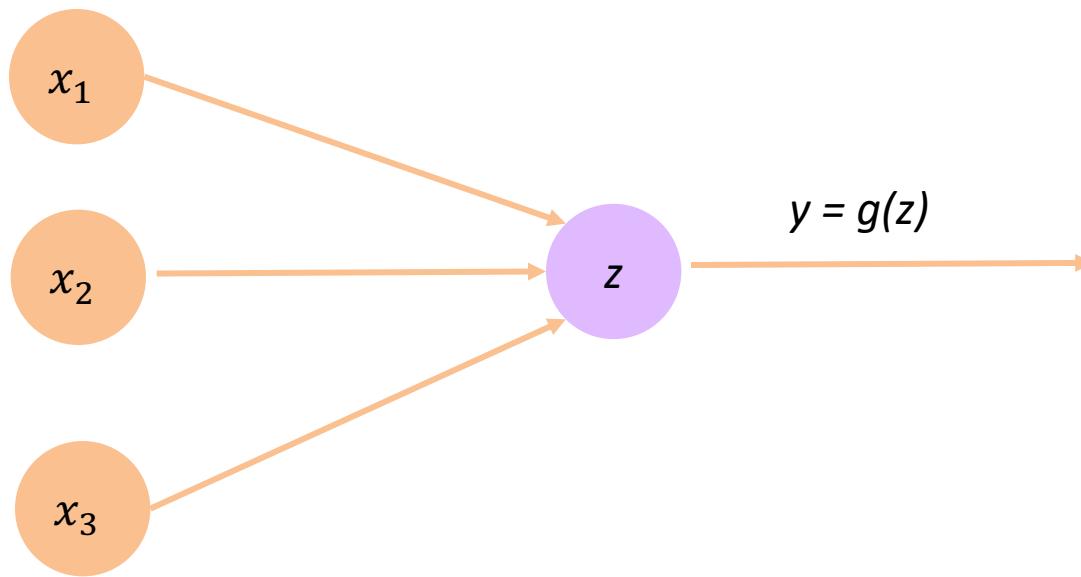


Building Neural Networks with Perceptrons

The Perceptron: Simplified



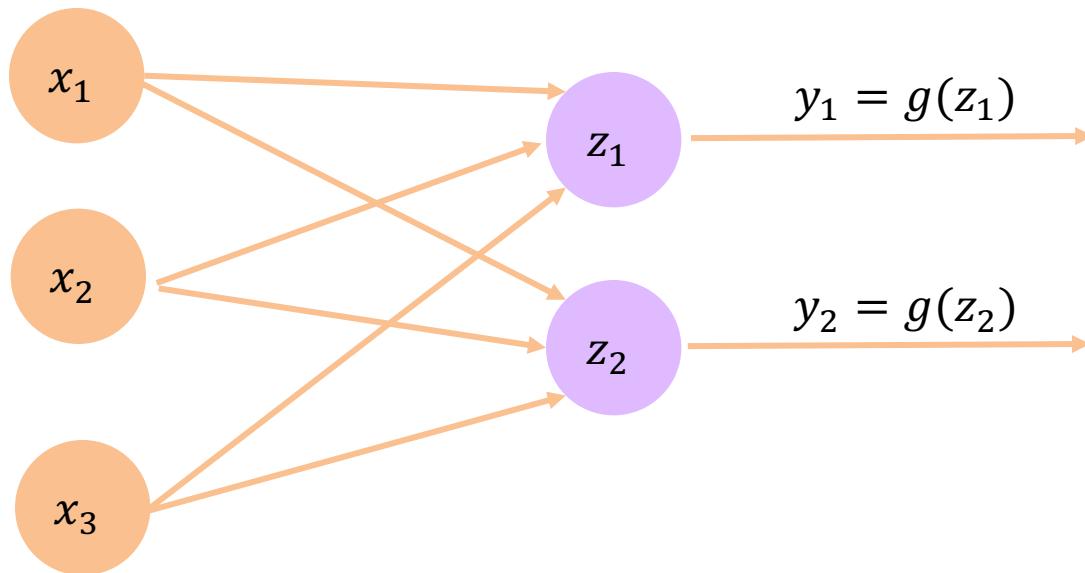
The Perceptron: Simplified



$$z = w_0 + \sum_{i=1}^3 x_i w_i$$

Multi Output Perceptron

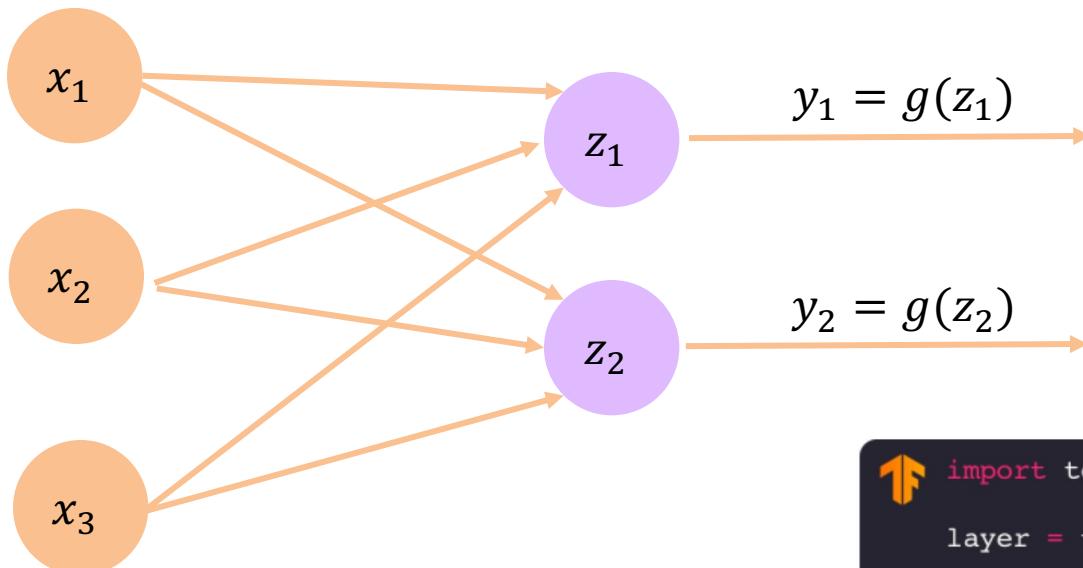
Because all inputs are densely connected to all outputs, these layers are called **Dense layers**



$$z_j = w_{0,j} + \sum_{i=1}^3 x_i w_{i,j}, \quad j = 1, 2$$

Multi Output Perceptron

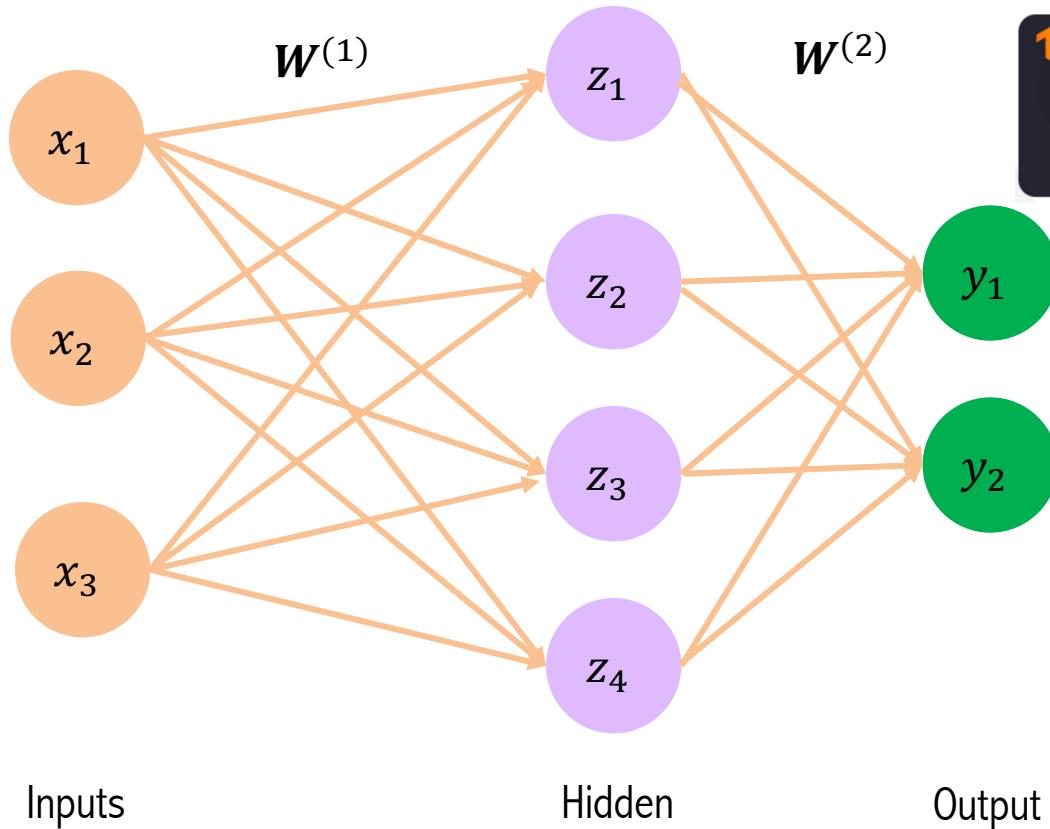
Because all inputs are densely connected to all outputs, these layers are called **Dense layers**



```
import tensorflow as tf  
  
layer = tf.keras.layers.Dense(  
    units=2)
```

$$z_j = w_{0,j} + \sum_{i=1}^3 x_i w_{i,j}, \quad j = 1, 2$$

Single (Hidden) Layer Neural Network

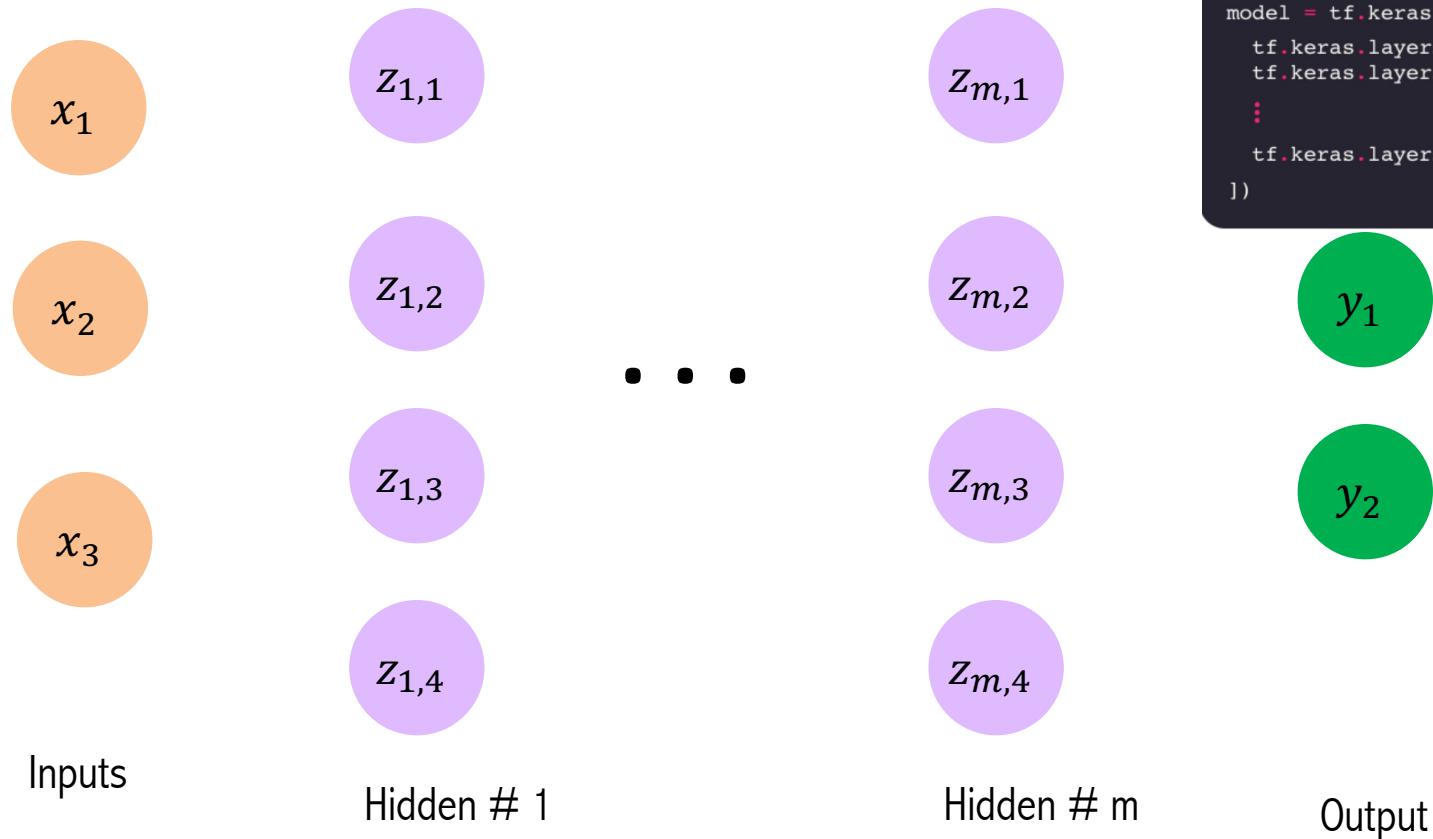


```
import tensorflow as tf  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(n),  
    tf.keras.layers.Dense(2)  
])
```

$$z_j = w_{0,j}^{(1)} + \sum_{i=1}^3 x_i w_{i,j}^{(1)}, \quad j = 1, 2, 3, 4.$$
$$y_k = w_{0,k}^{(2)} + \sum_{j=1}^4 g(z_j) w_{j,k}^{(2)}, \quad k = 1, 2.$$

Deep Neural Networks

"Deep" means more than 1 hidden layer



```
TensorFlow logo  
import tensorflow as tf  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(n1),  
    tf.keras.layers.Dense(n2),  
    ...  
    tf.keras.layers.Dense(2)  
])
```

Applying Neural Networks

Example Problem

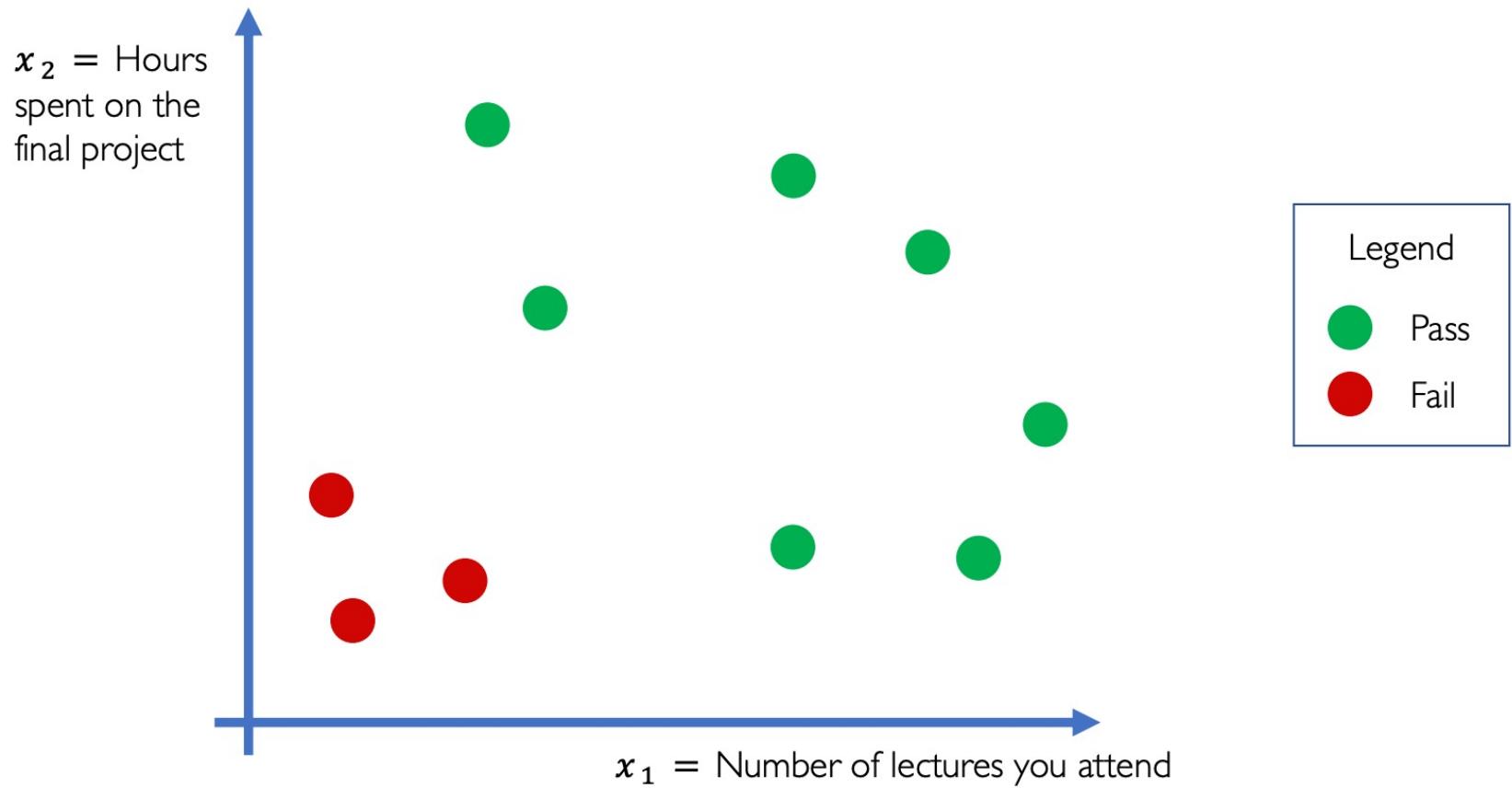
Will I pass this class?

Let's start with a simple two feature model

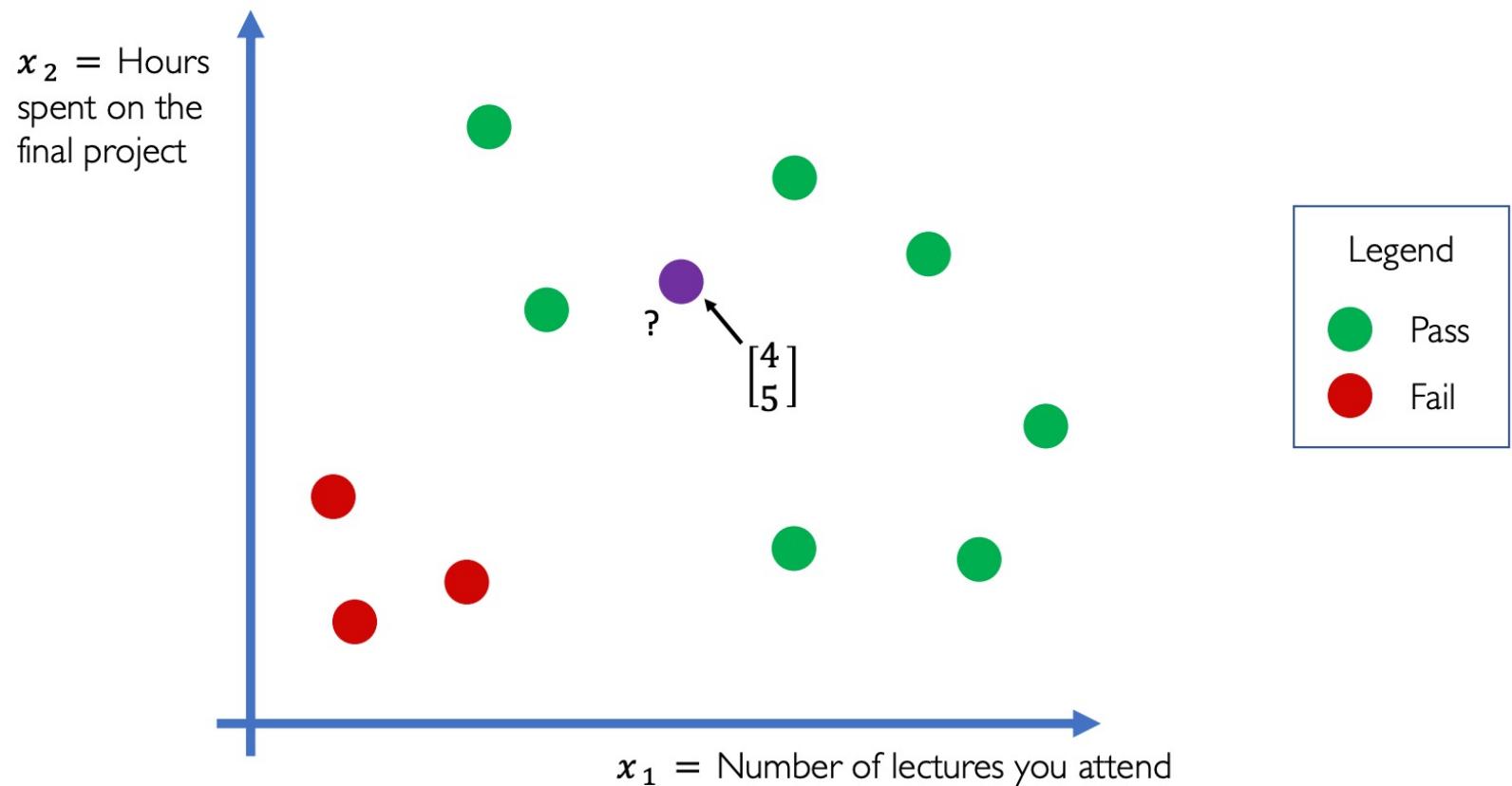
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

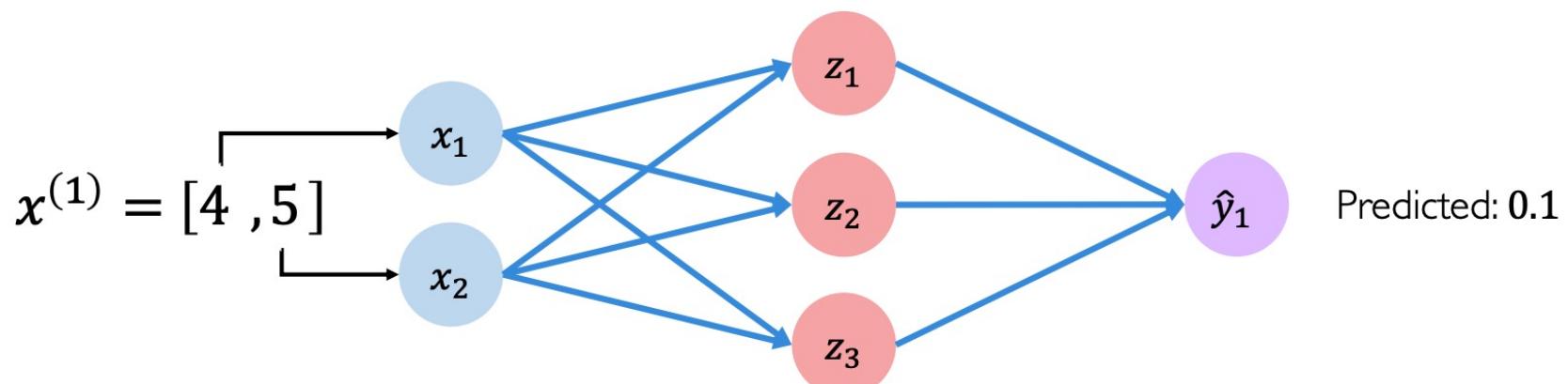
Example Problem: Will I pass this class?

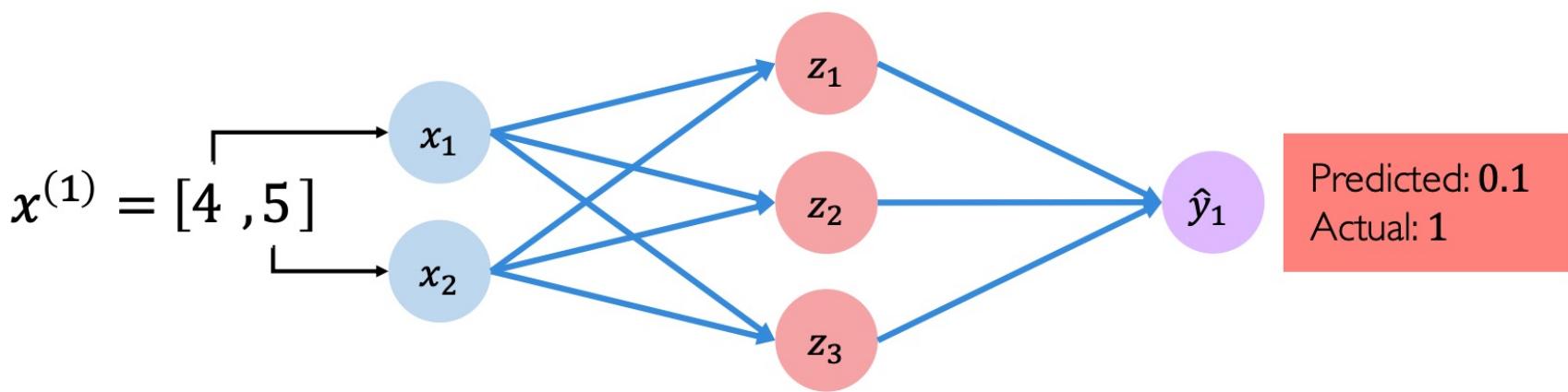


Example Problem: Will I pass this class?



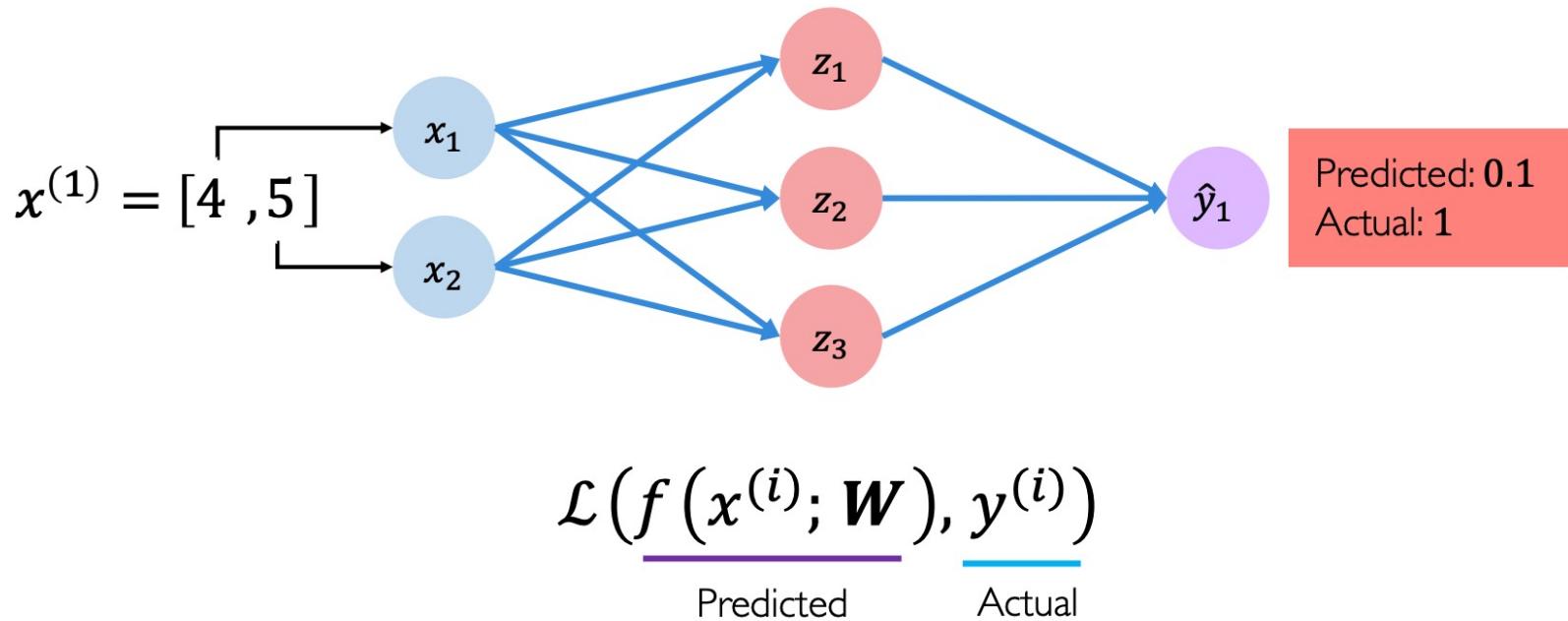
Example Problem: Will I pass this class?





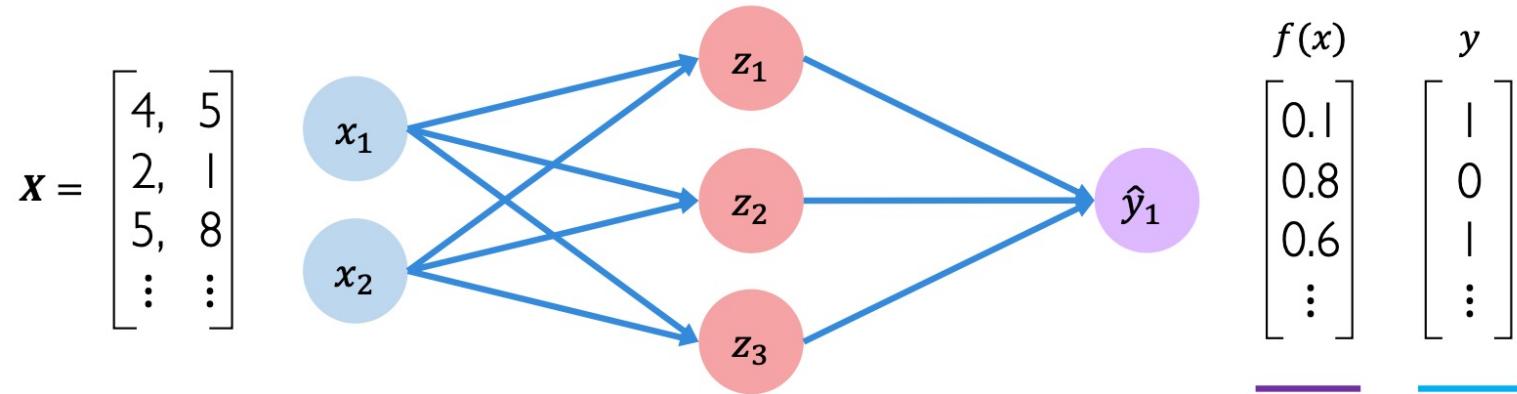
Quantifying Loss

The **loss** of our network measures the cost incurred from incorrect predictions



Empirical Loss

The **empirical loss** measures the total loss over our entire dataset



Also known as:

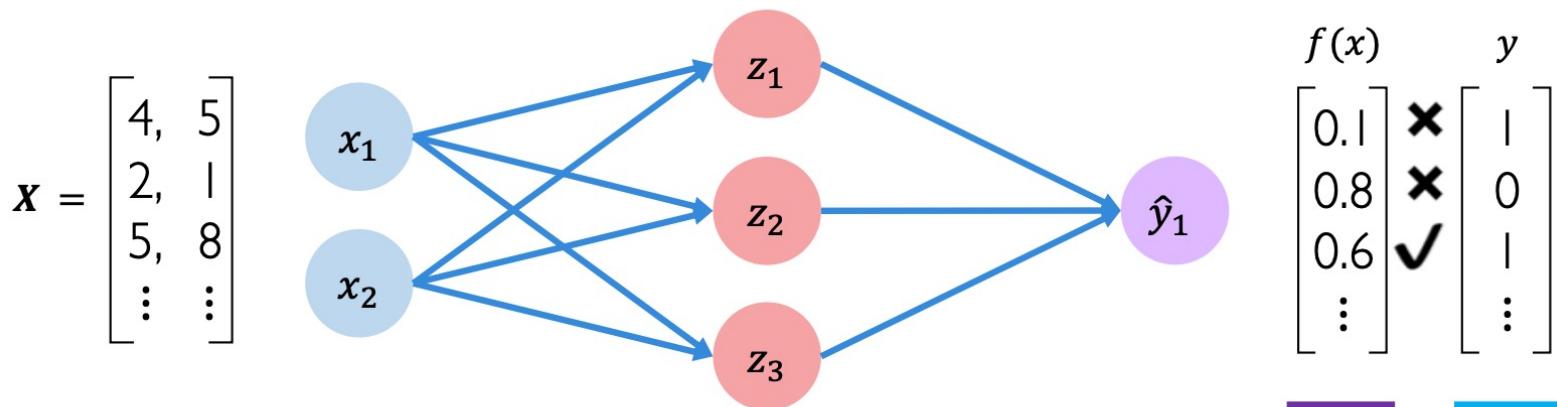
- Objective function
- Cost function
- Empirical Risk

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

Predicted Actual

Binary Cross Entropy Loss

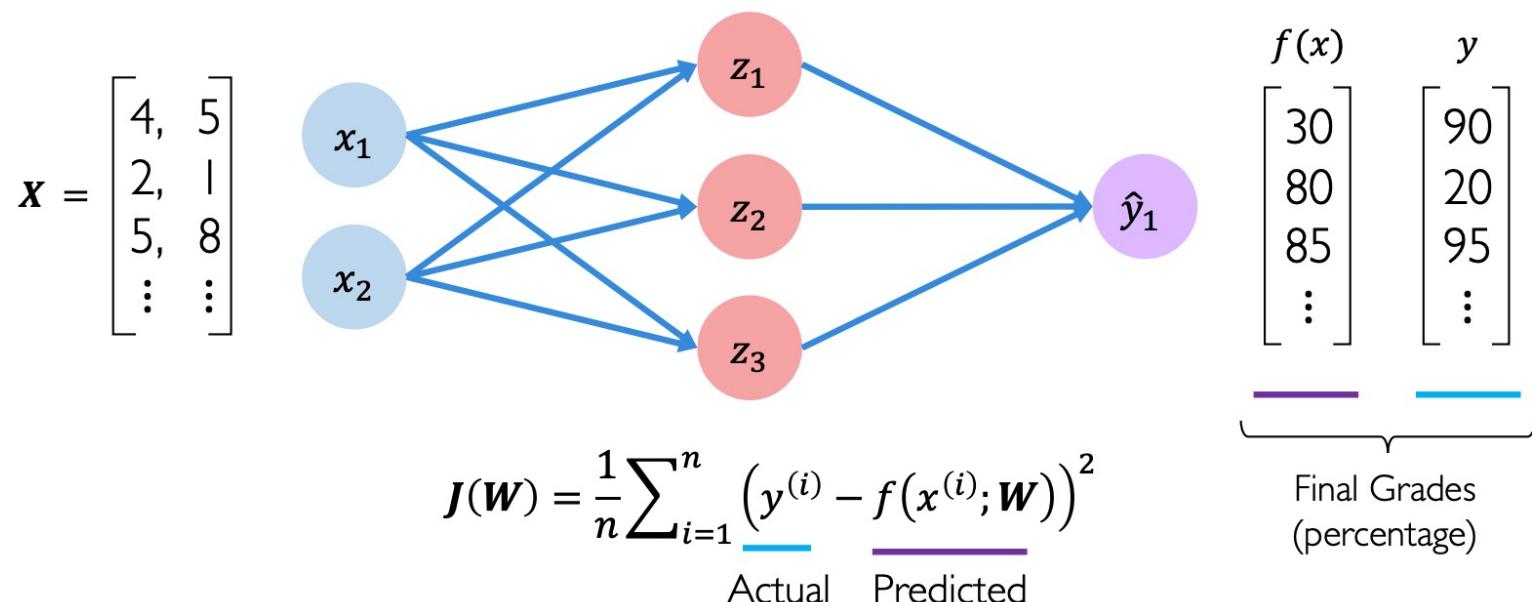
Cross entropy loss can be used with models that output a probability between 0 and 1



```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
```

Training Neural Networks

Typical training procedure for a neural network

1. Define the neural network that has some learnable parameters (or weights)
2. Iterate over a dataset of inputs
3. Process input through the network
4. Compute the loss (how far is the output from being correct)
5. Propagate gradients back into the network's parameters
6. Update the weights of the network, typically using a simple update rule: $weight = weight - learning_rate * gradient$

Loss Optimization

We want to find the network weights that **achieve the lowest loss**

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

Loss Optimization

We want to *find the network weights that achieve the lowest loss*

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

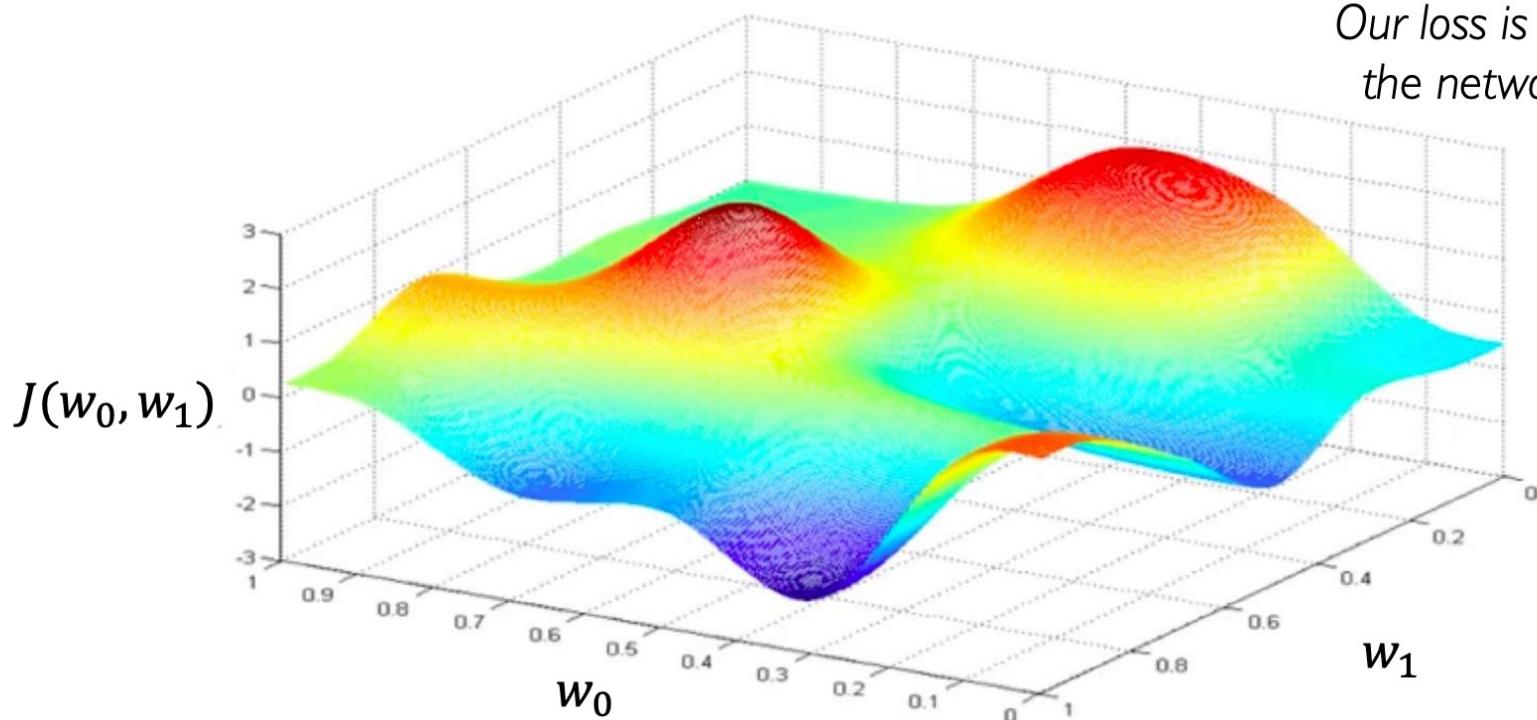
$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$



Remember:
 $\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \dots\}$

Loss Optimization

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

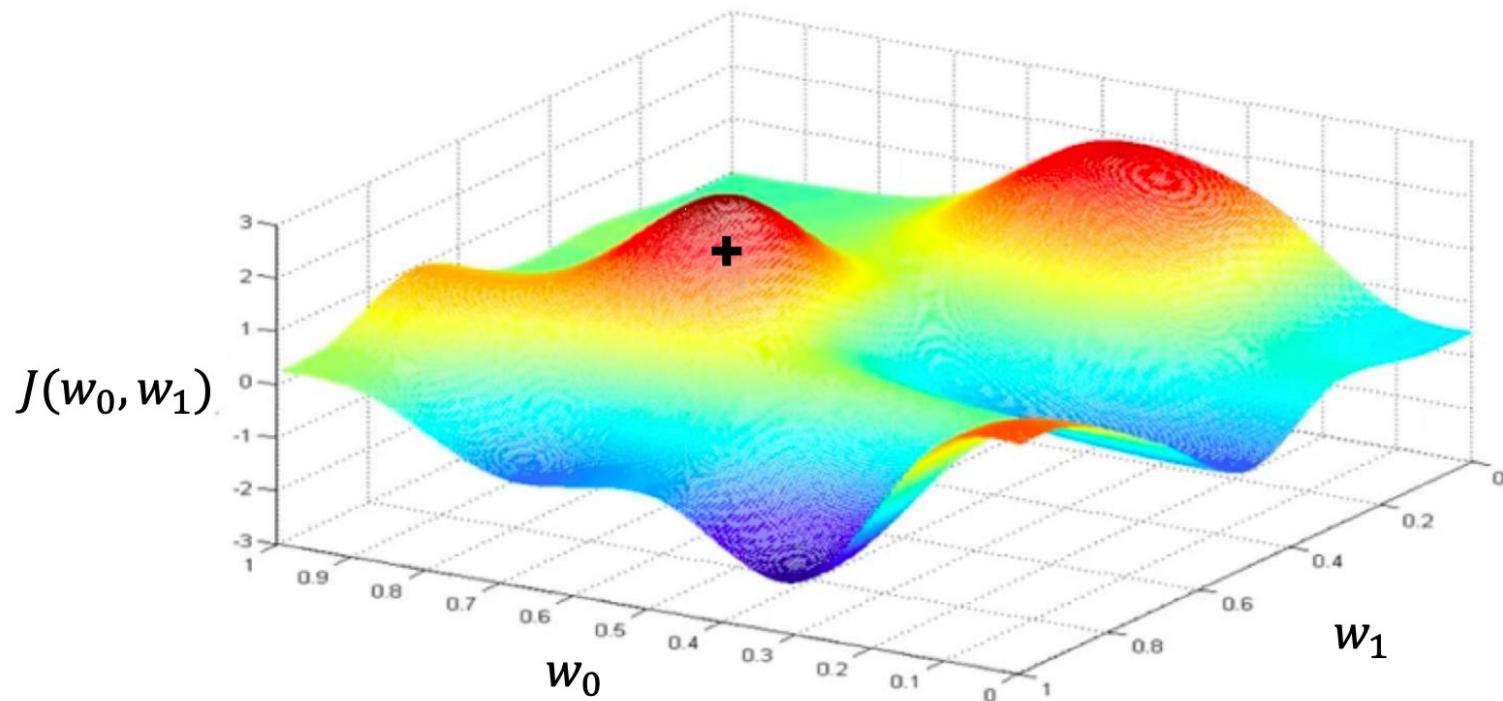


Remember:

*Our loss is a function of
the network weights!*

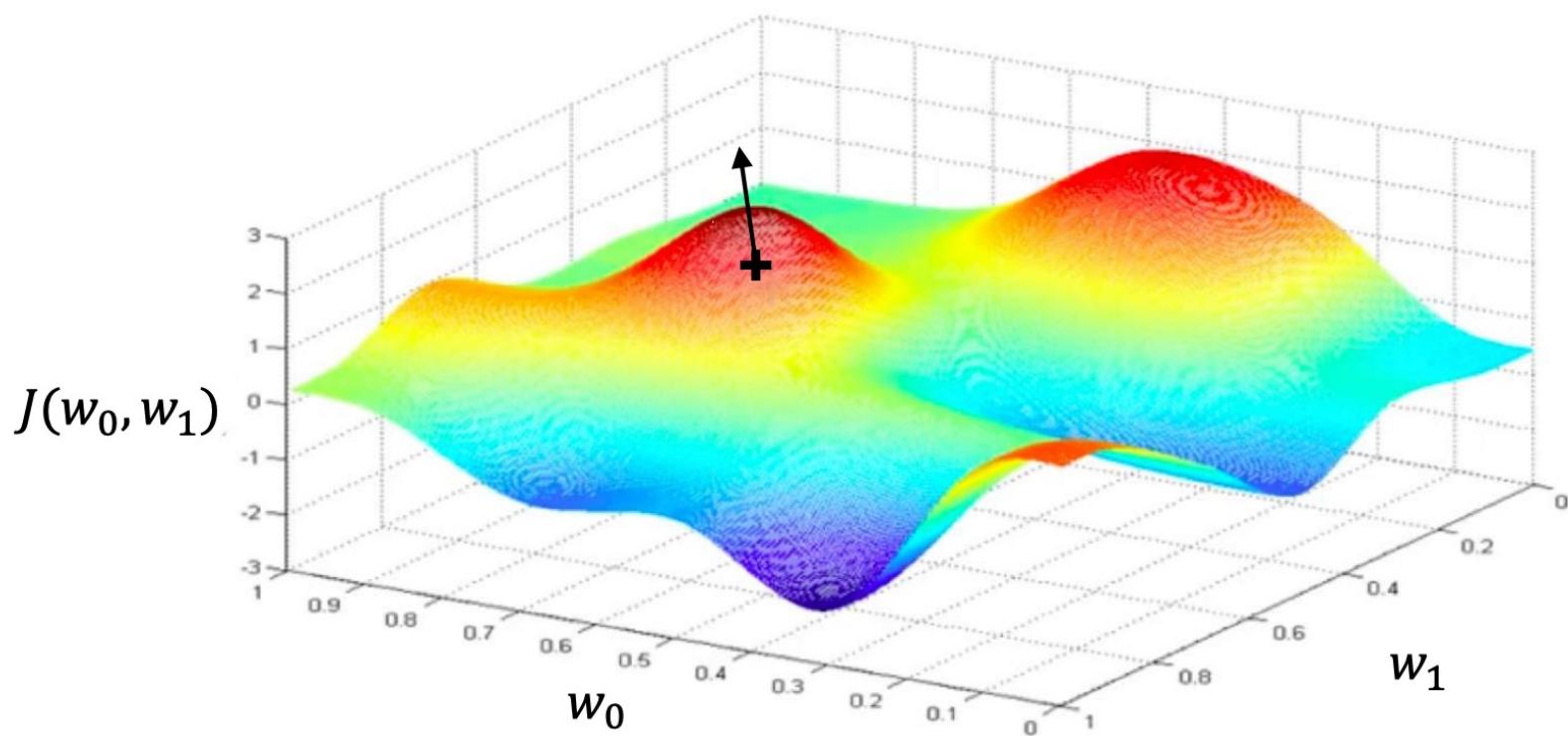
Loss Optimization

Randomly pick an initial (w_0, w_1)



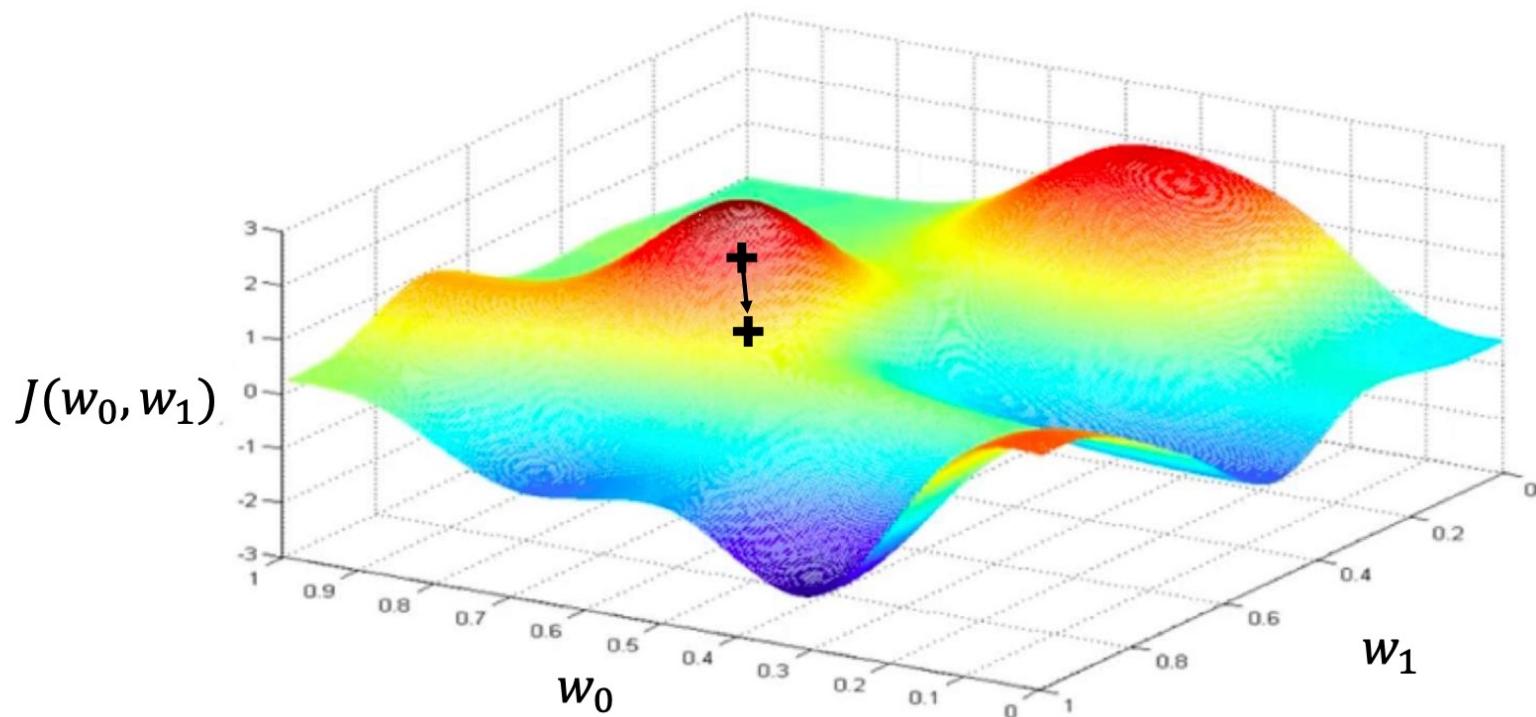
Loss Optimization

Compute gradient, $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$



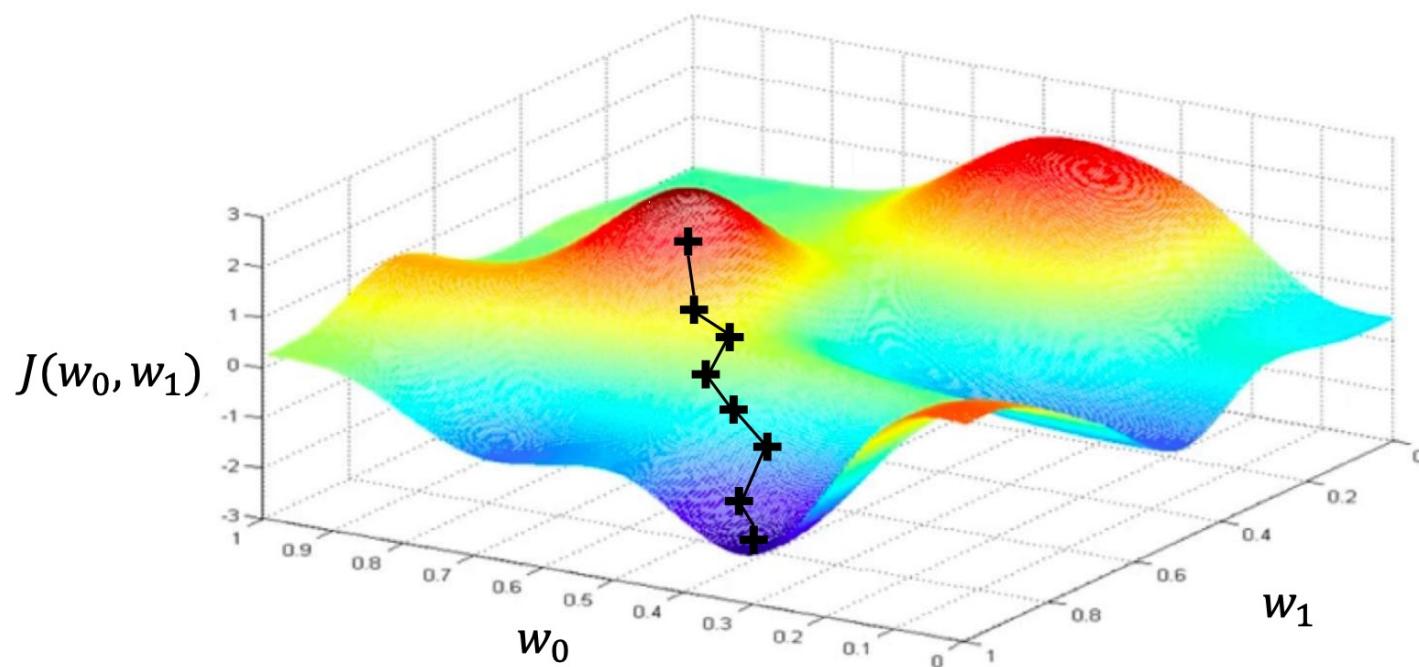
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

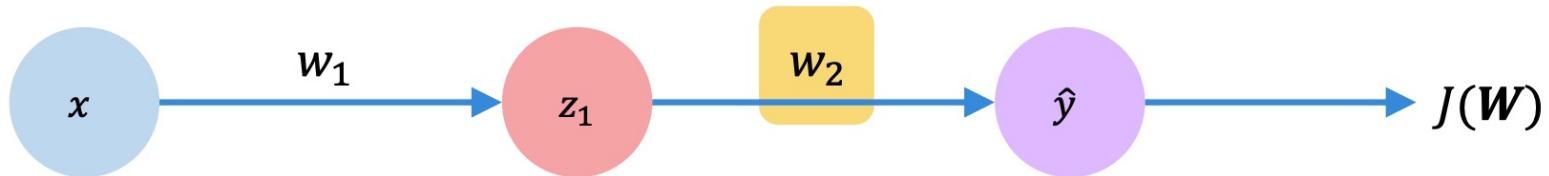
```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

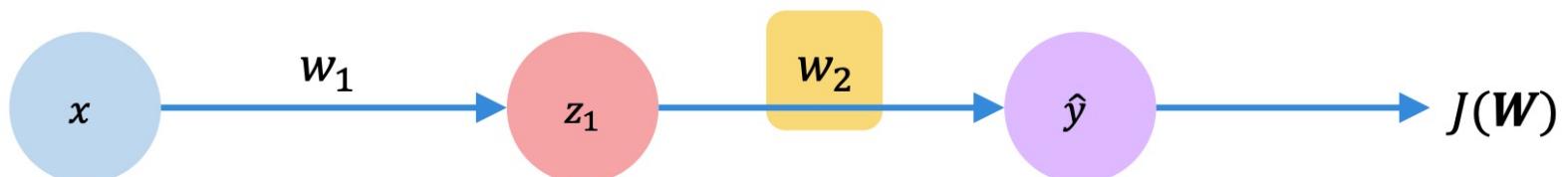
    weights = weights - lr * gradient
```

Computing Gradients: Backpropagation



How does a small change in one weight (ex. w_2) affect the final loss $J(\mathbf{W})$?

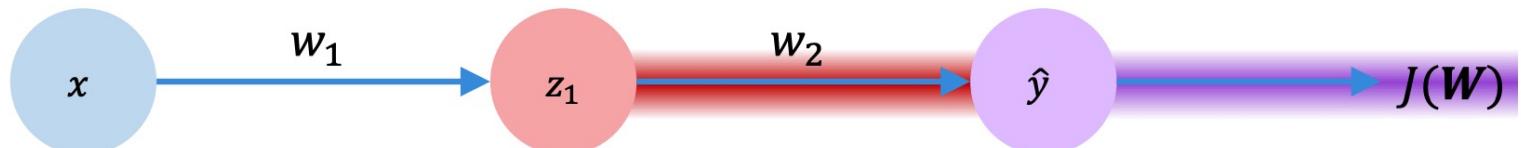
Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_2} =$$

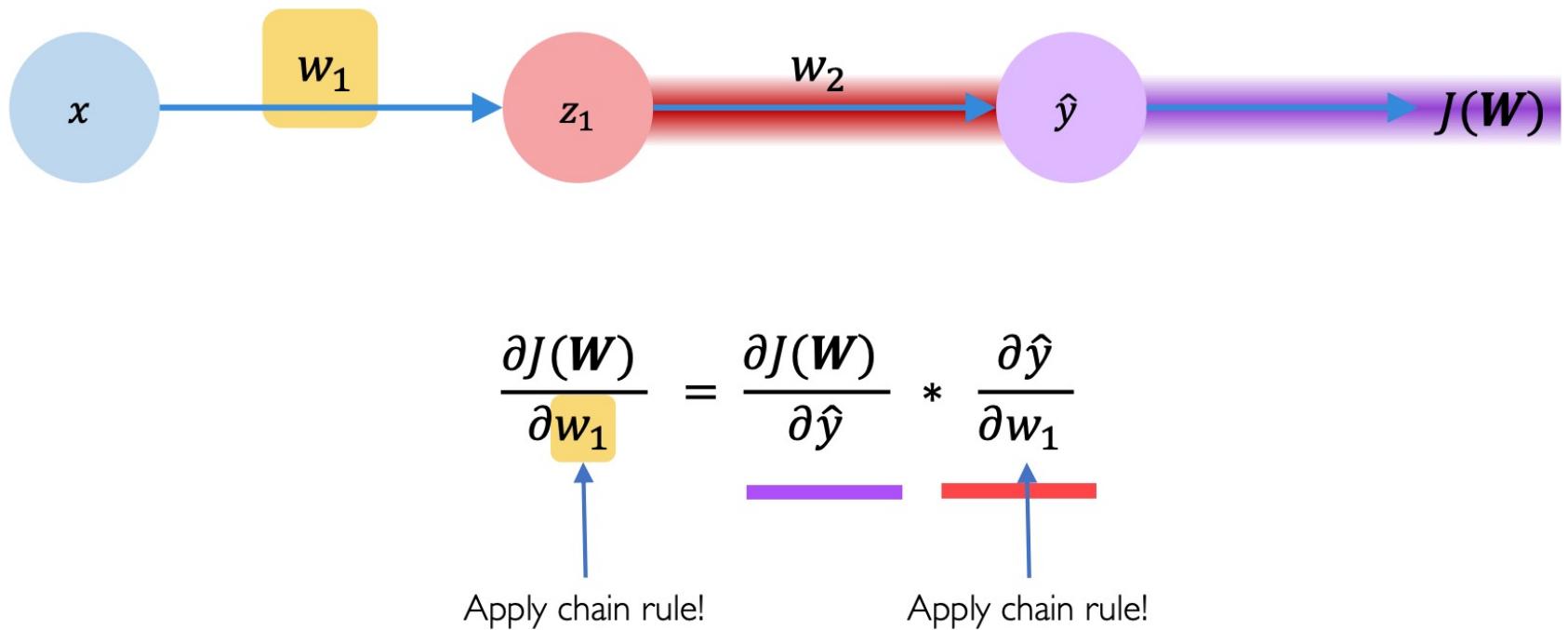
Let's use the chain rule!

Computing Gradients: Backpropagation

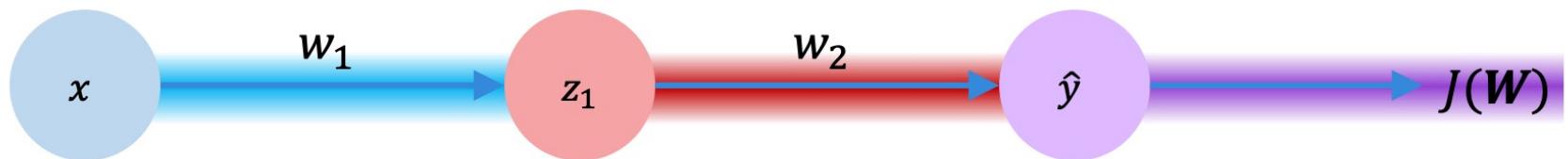


$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \underline{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}} * \overline{\frac{\partial \hat{y}}{\partial w_2}}$$

Computing Gradients: Backpropagation

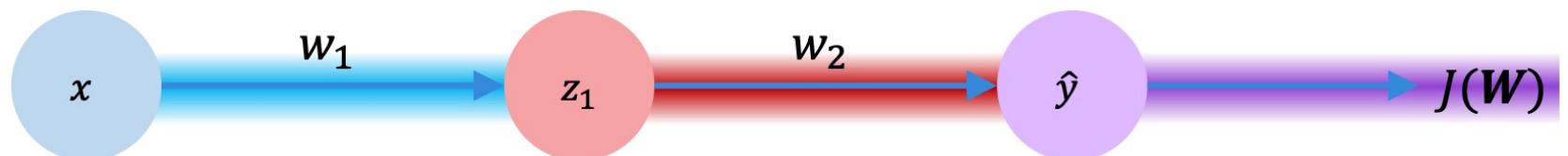


Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple bar}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red bar}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue bar}}$$

Computing Gradients: Backpropagation

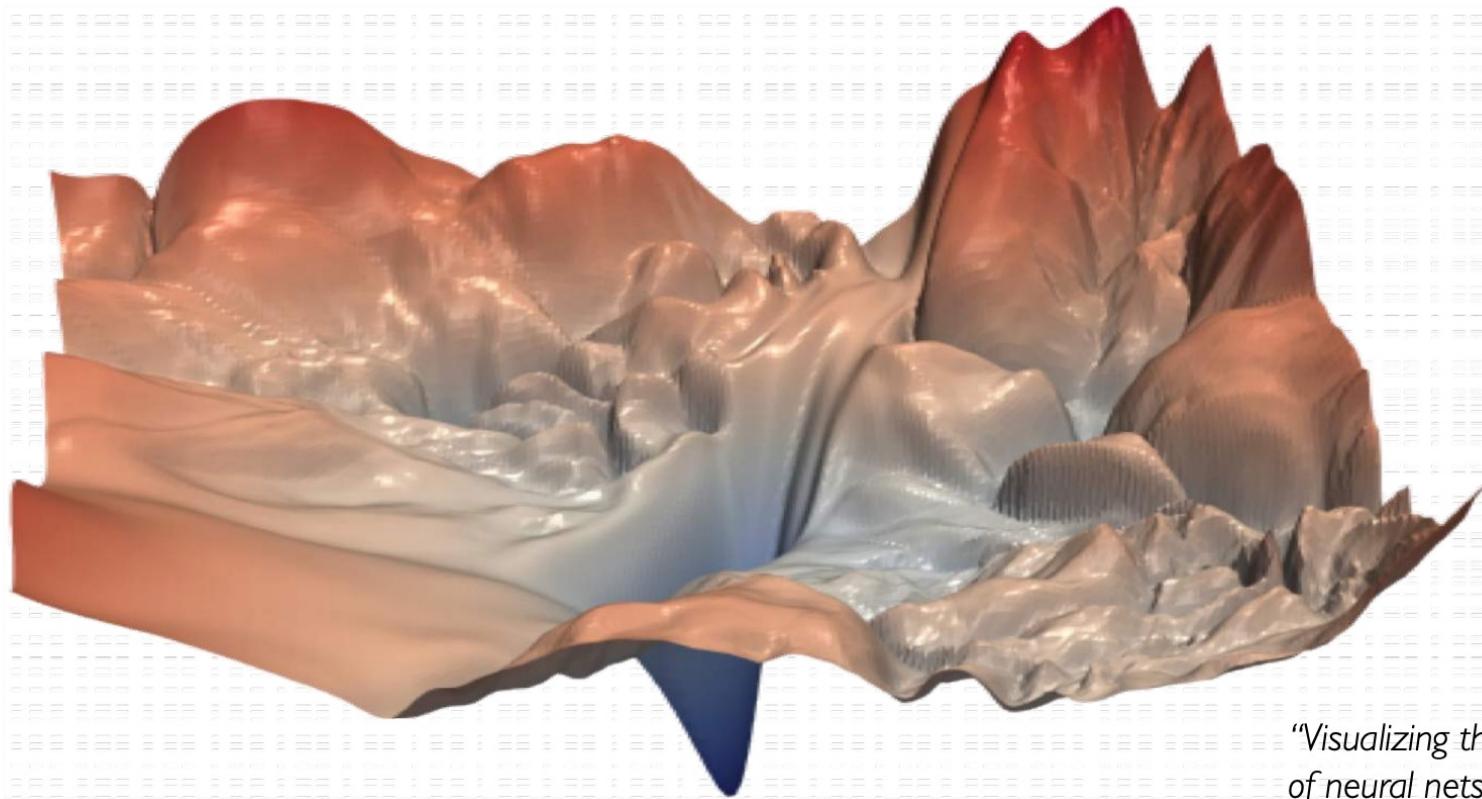


$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

Repeat this for **every weight in the network** using gradients from later layers

Neural Networks in Practice: Optimization

Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Loss Functions Can Be Difficult to Optimize

Remember:

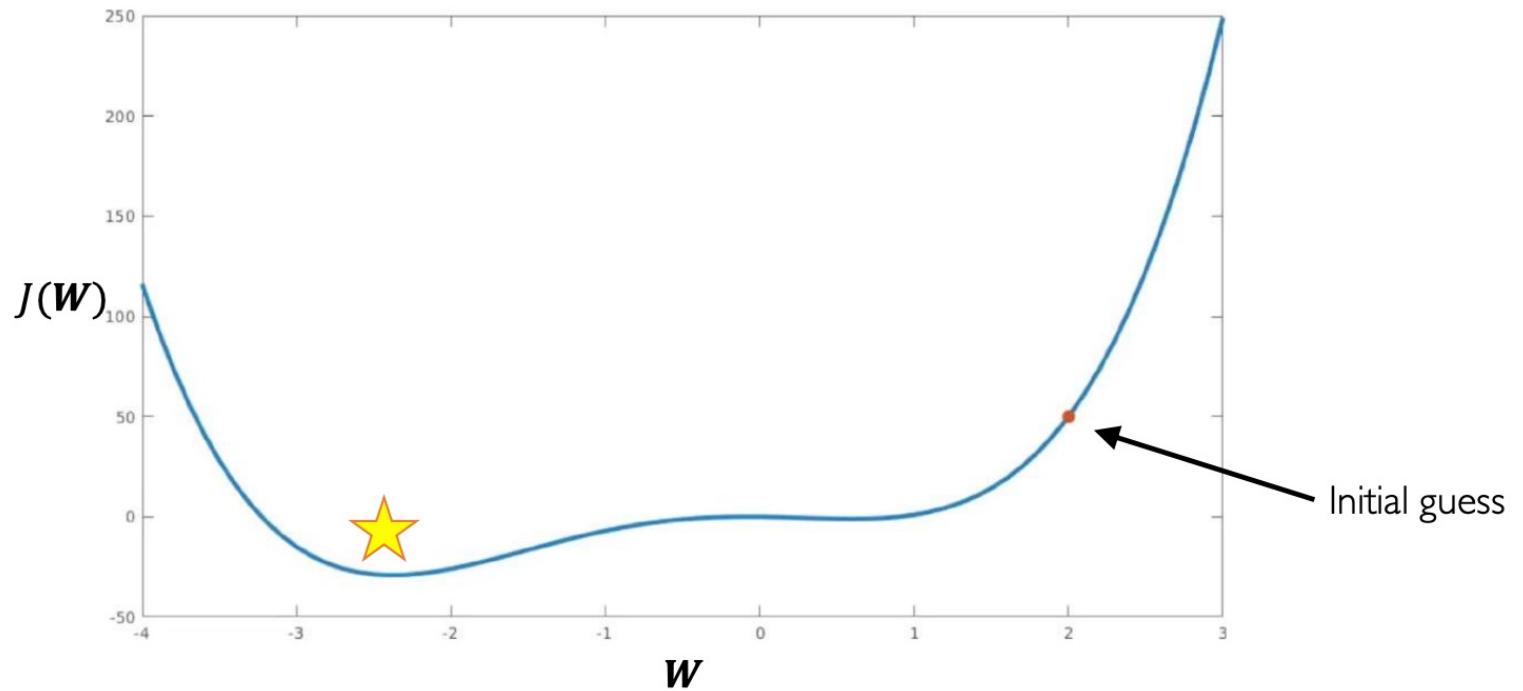
Optimization through gradient descent

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

How can we set the
learning rate?

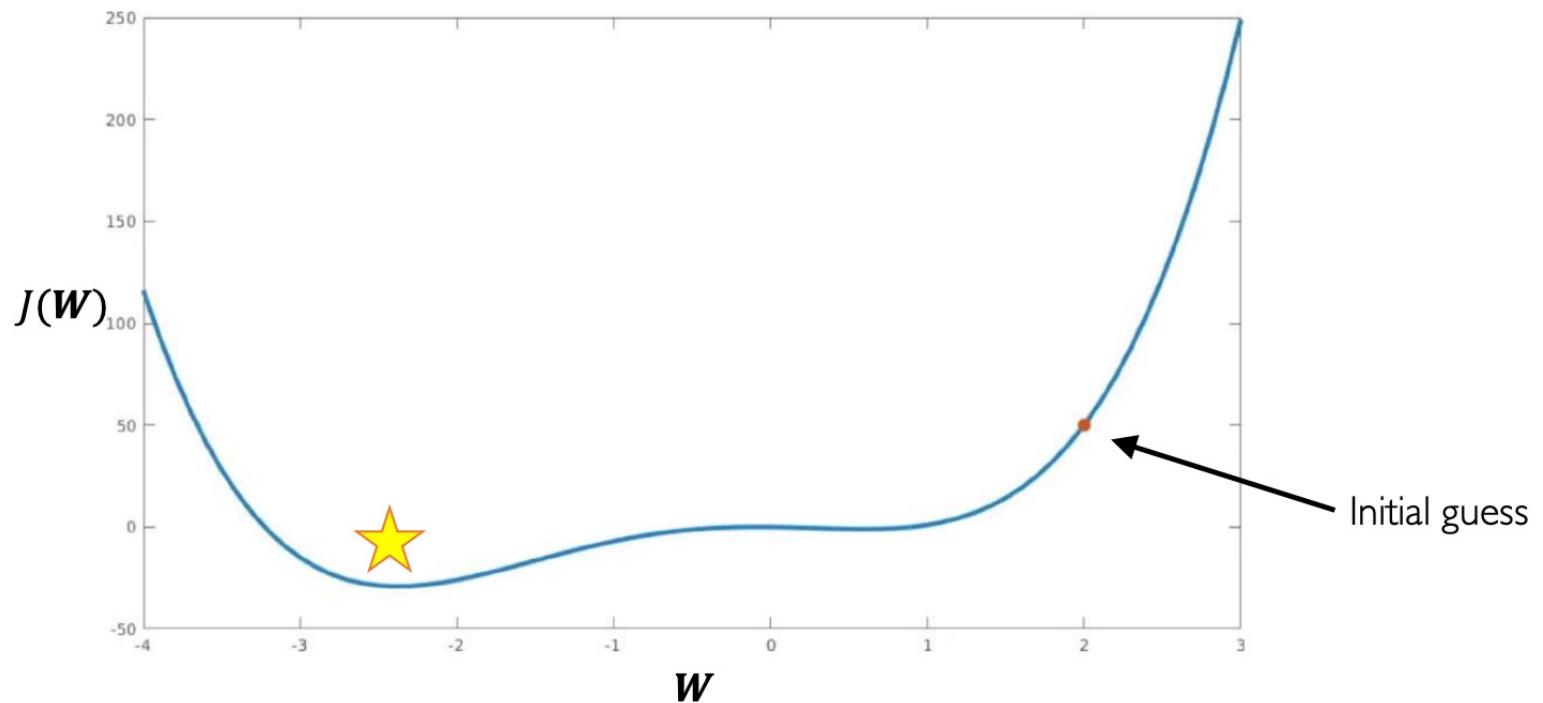
Setting the Learning Rate

Small learning rate converges slowly and gets stuck in false local minima



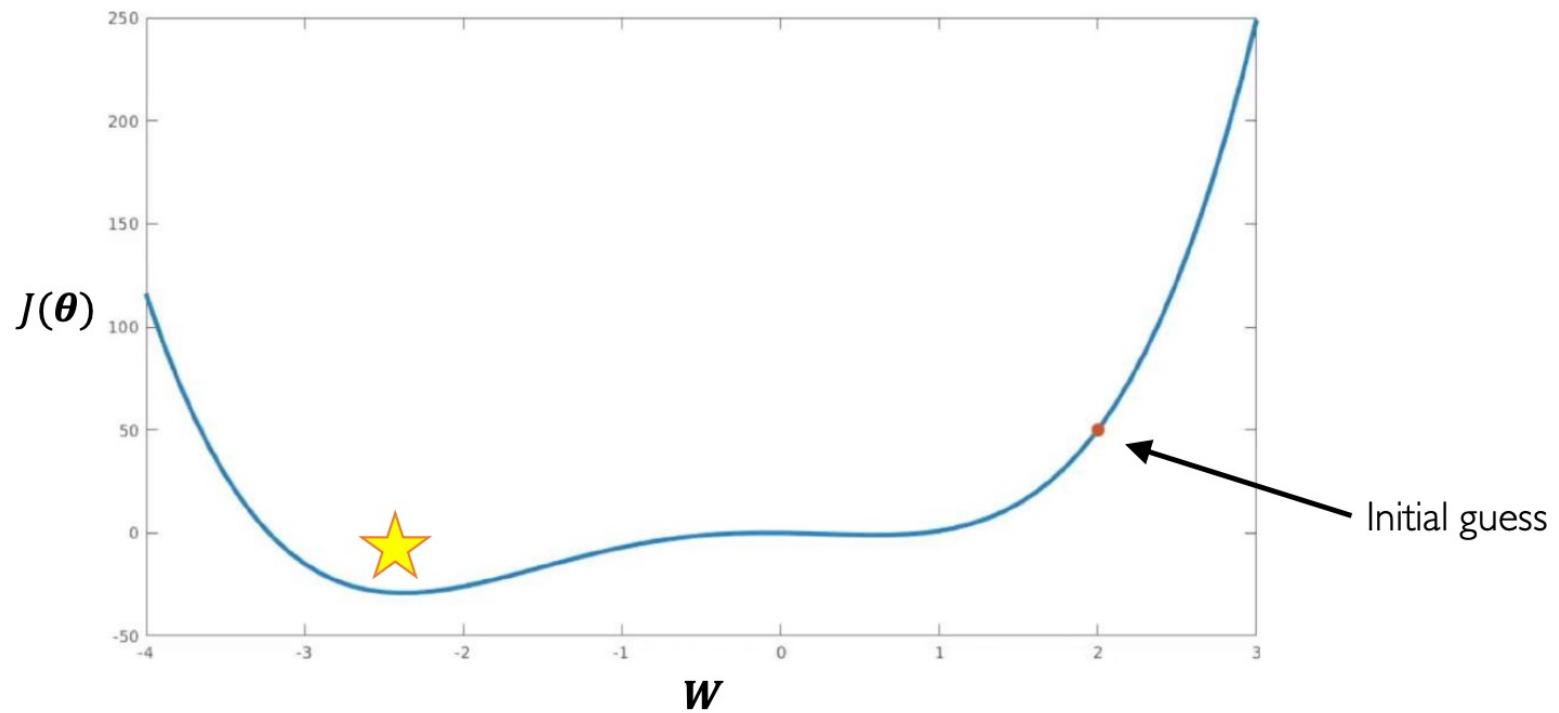
Setting the Learning Rate

Large learning rates overshoot, become unstable and diverge



Setting the Learning Rate

Stable learning rates converge smoothly and avoid local minima



How to deal with this?

Idea I:

Try lots of different learning rates and see what works “just right”

How to deal with this?

Idea 1:

Try lots of different learning rates and see what works “just right”

Idea 2:

Do something smarter!

Design an adaptive learning rate that “adapts” to the landscape

Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
 - how large gradient is
 - how fast learning is happening
 - size of particular weights
 - etc...

Adaptive Learning Rate Algorithms

Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

TF Implementation



`tf.keras.optimizers.SGD`



`tf.keras.optimizers.Adam`



`tf.keras.optimizers.Adadelta`



`tf.keras.optimizers.Adagrad`



`tf.keras.optimizers.RMSProp`

Reference

Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Additional details: <http://ruder.io/optimizing-gradient-descent/>

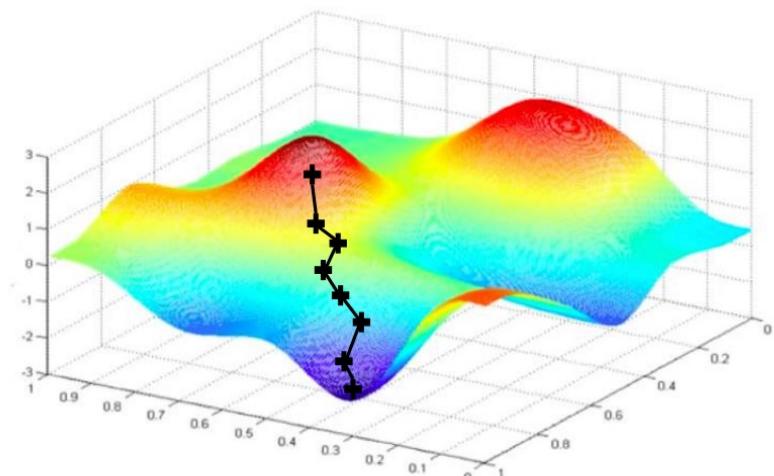
Neural Networks in Practice: Mini-batches

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

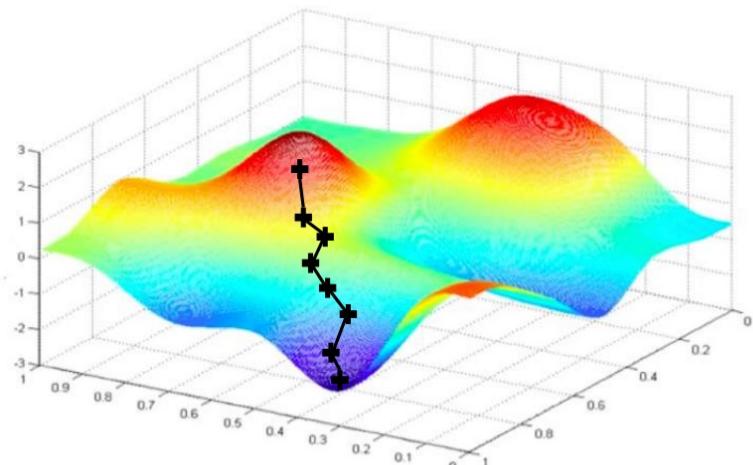
Can be very
computational to
compute!



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights

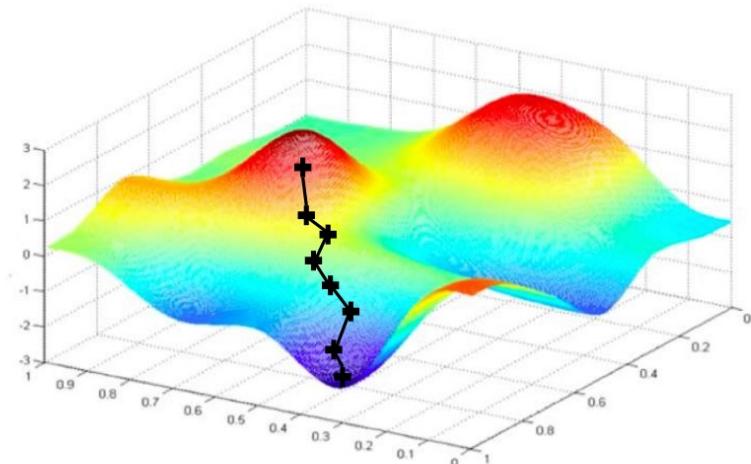


Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights

Easy to compute but
very noisy
(stochastic)!

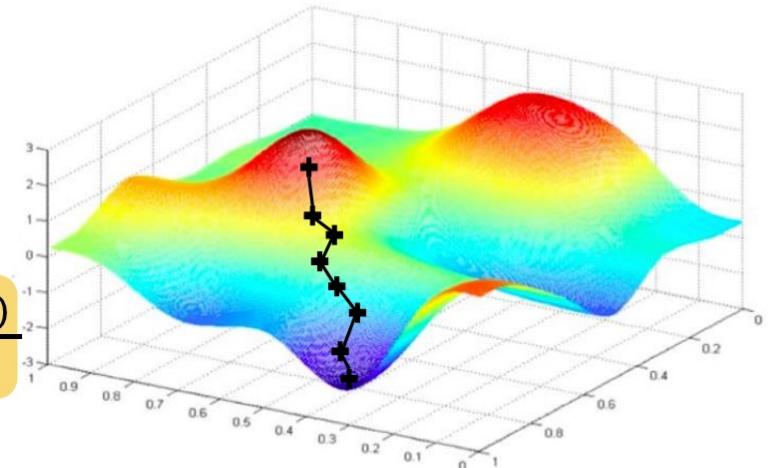


Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient,
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights

Fast to compute and a much better estimate of the true gradient!



Stochastic Gradient Descent

- Epochs
 - One Epoch is when an **ENTIRE** dataset is passed forward and backward through the neural network only **ONCE**
 - Updating the weights with single pass or one epoch is not enough
 - As the number of epochs increases, more number of times the weights are changed in neural network (underfitting -> optimal -> overfitting)
- Batch Size
 - Total number of training examples present in a single batch
 - Since one epoch is too big to feed to the computer at once we divide it in several smaller batches
- Iterations
 - Iterations is the number of batches needed to complete one epoch
- E.g., we can divide the dataset of 2000 examples into batches of 500 then it will take 4 iterations to complete 1 epoch.

Additional details: <https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9>

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

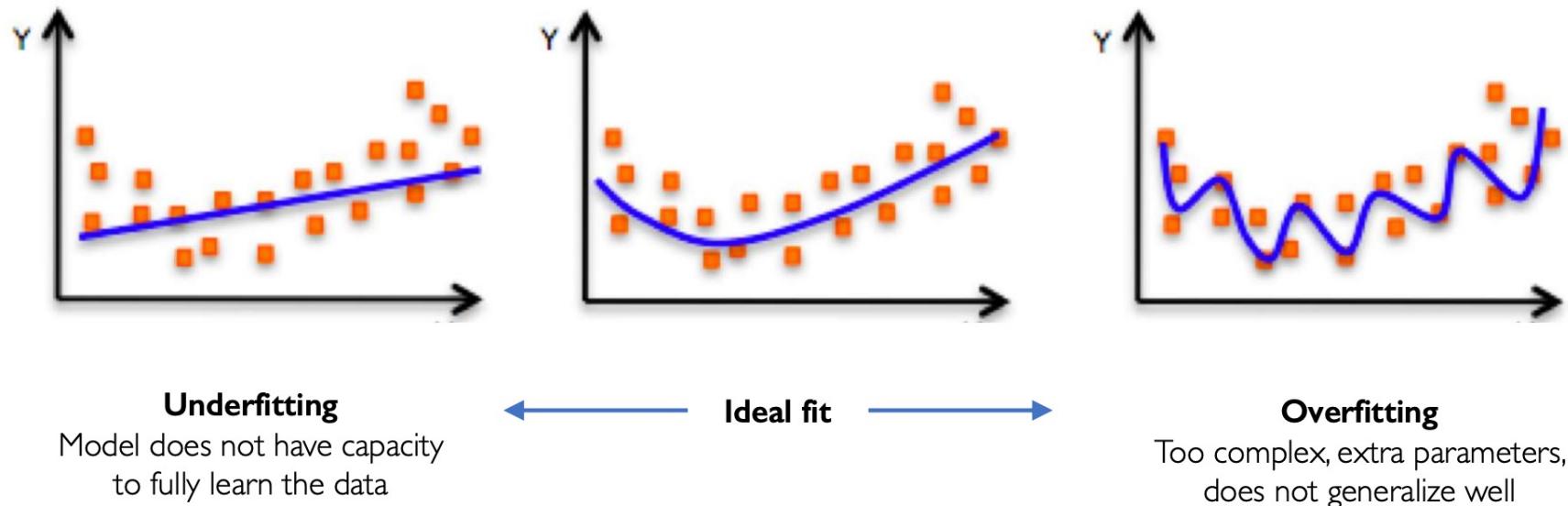
Allows for larger learning rates

Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

Neural Networks in Practice: Overfitting

The Problem of Overfitting



Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

Regularization

What is it?

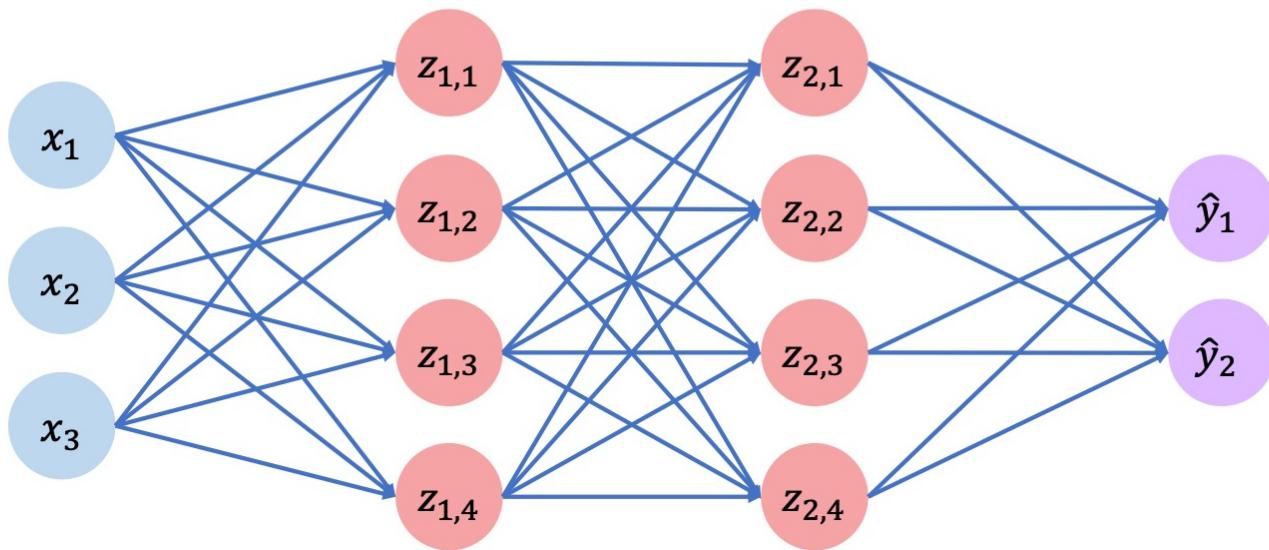
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Regularization I: Dropout

- During training, randomly set some activations to 0

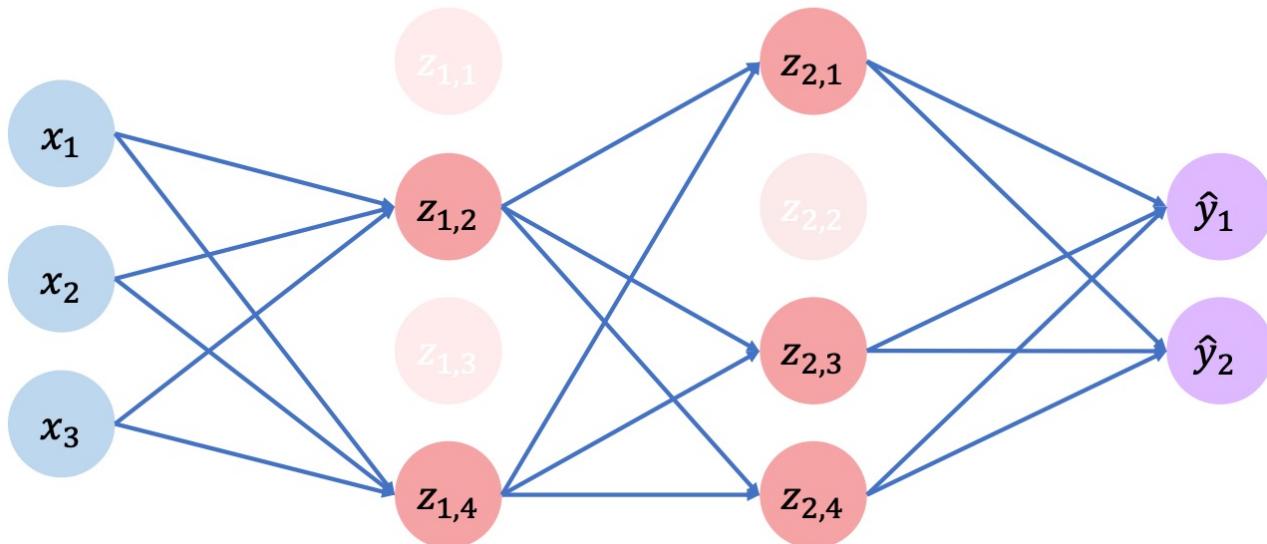


Regularization I: Dropout

- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node



`tf.keras.layers.Dropout (p=0.5)`

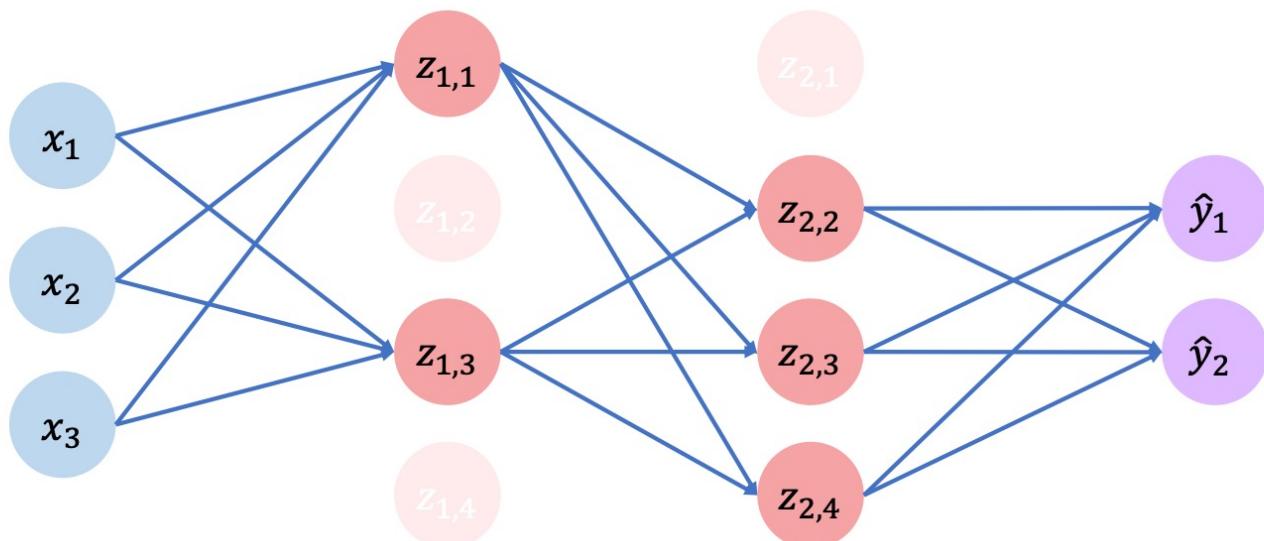


Regularization I: Dropout

- During training, randomly set some activations to 0
 - Typically ‘drop’ 50% of activations in layer
 - Forces network to not rely on any 1 node



`tf.keras.layers.Dropout(p=0.5)`



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



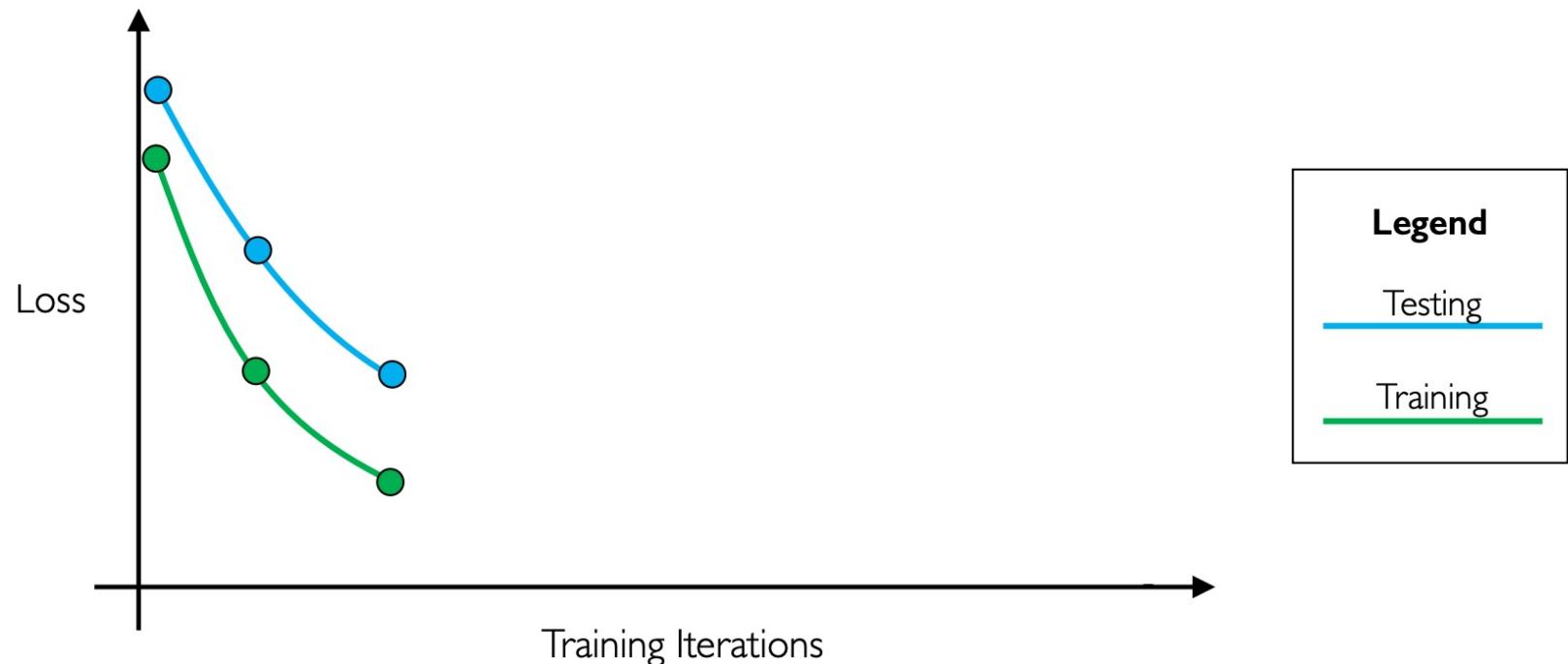
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



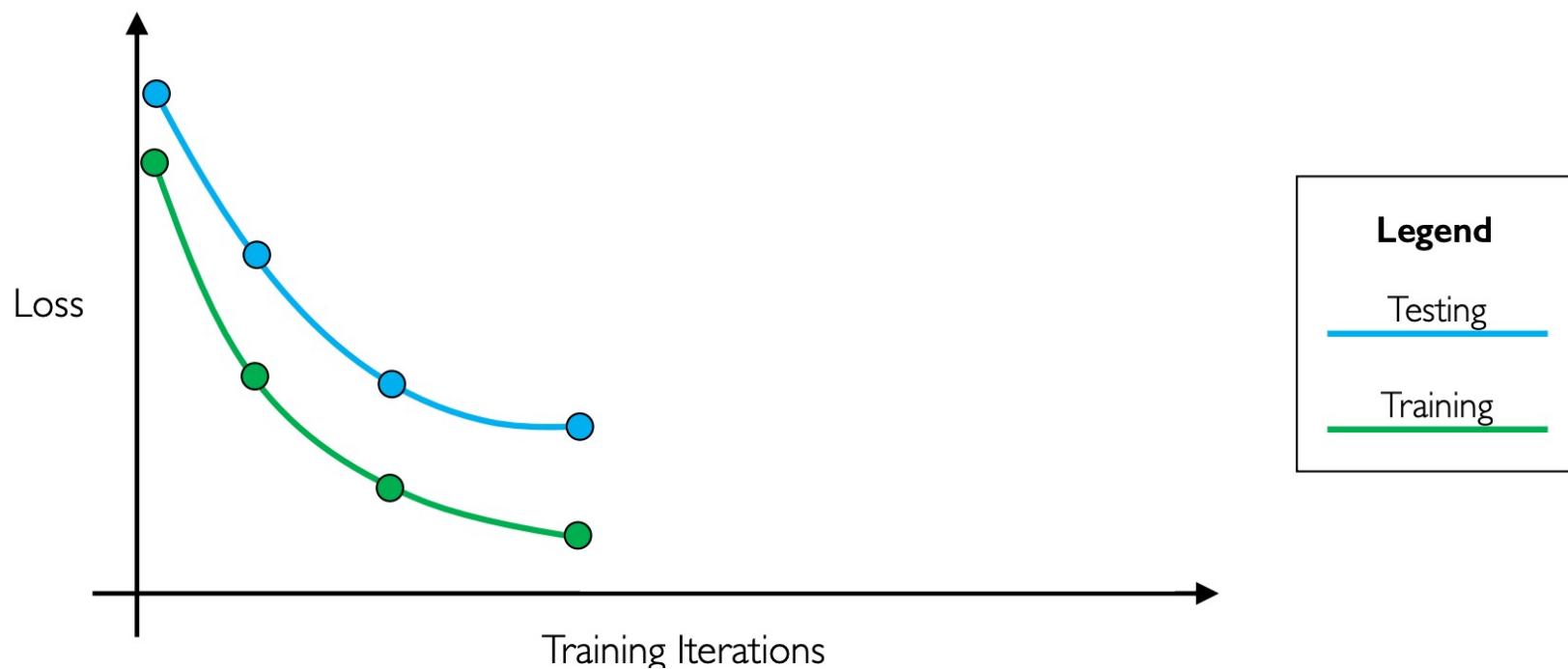
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



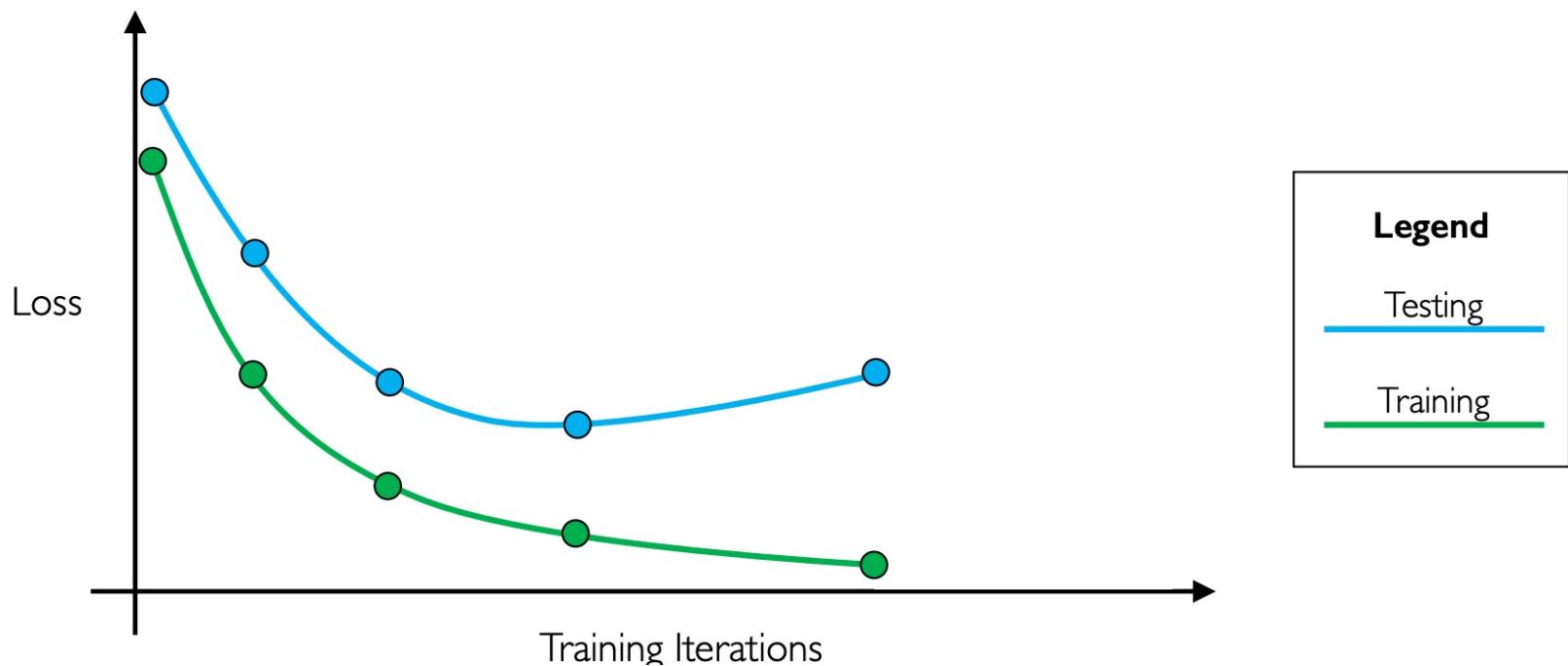
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



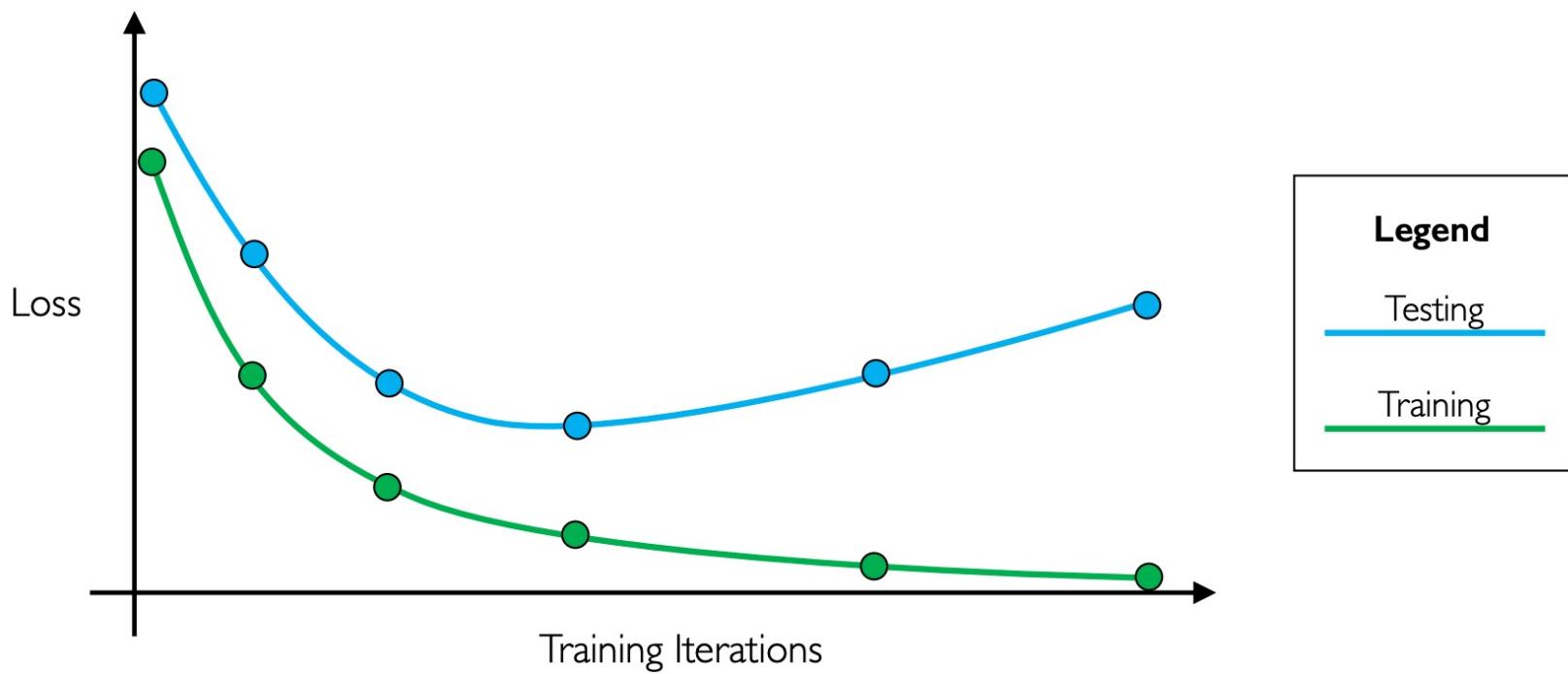
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



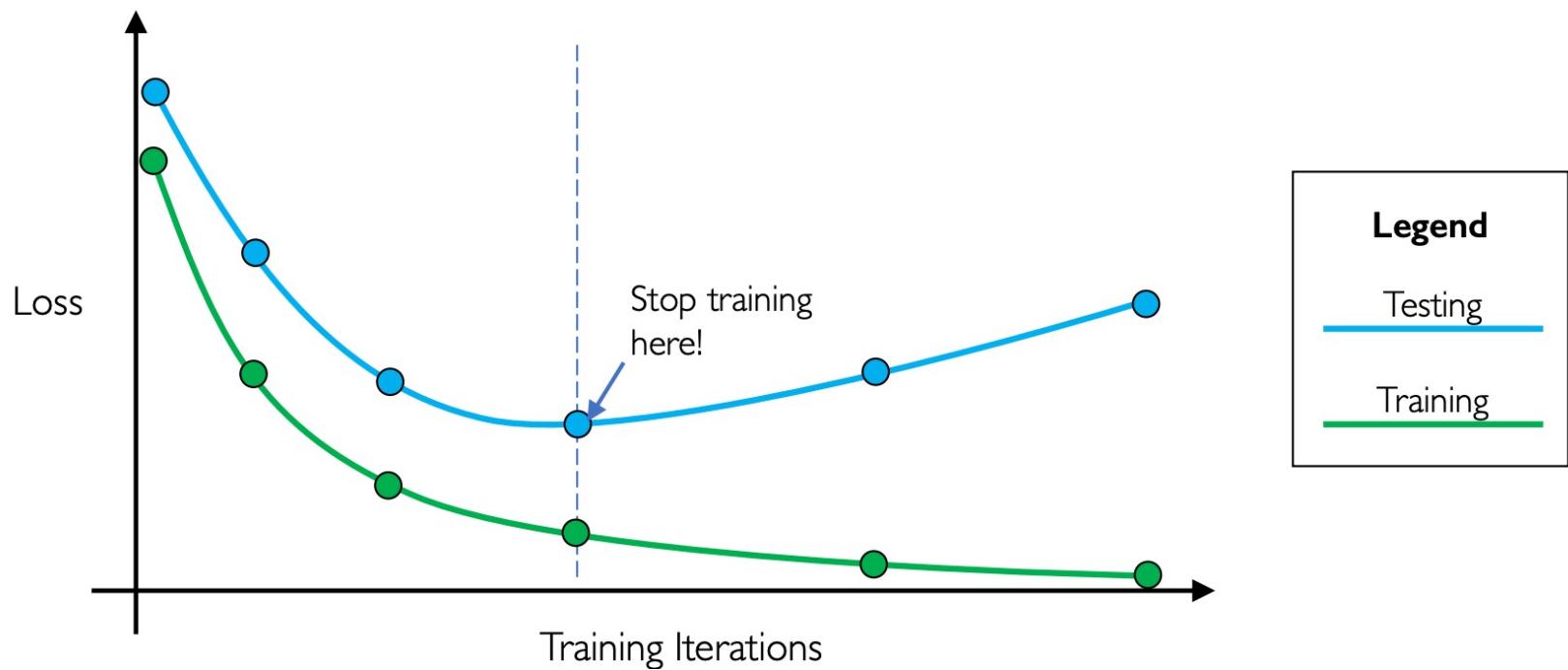
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



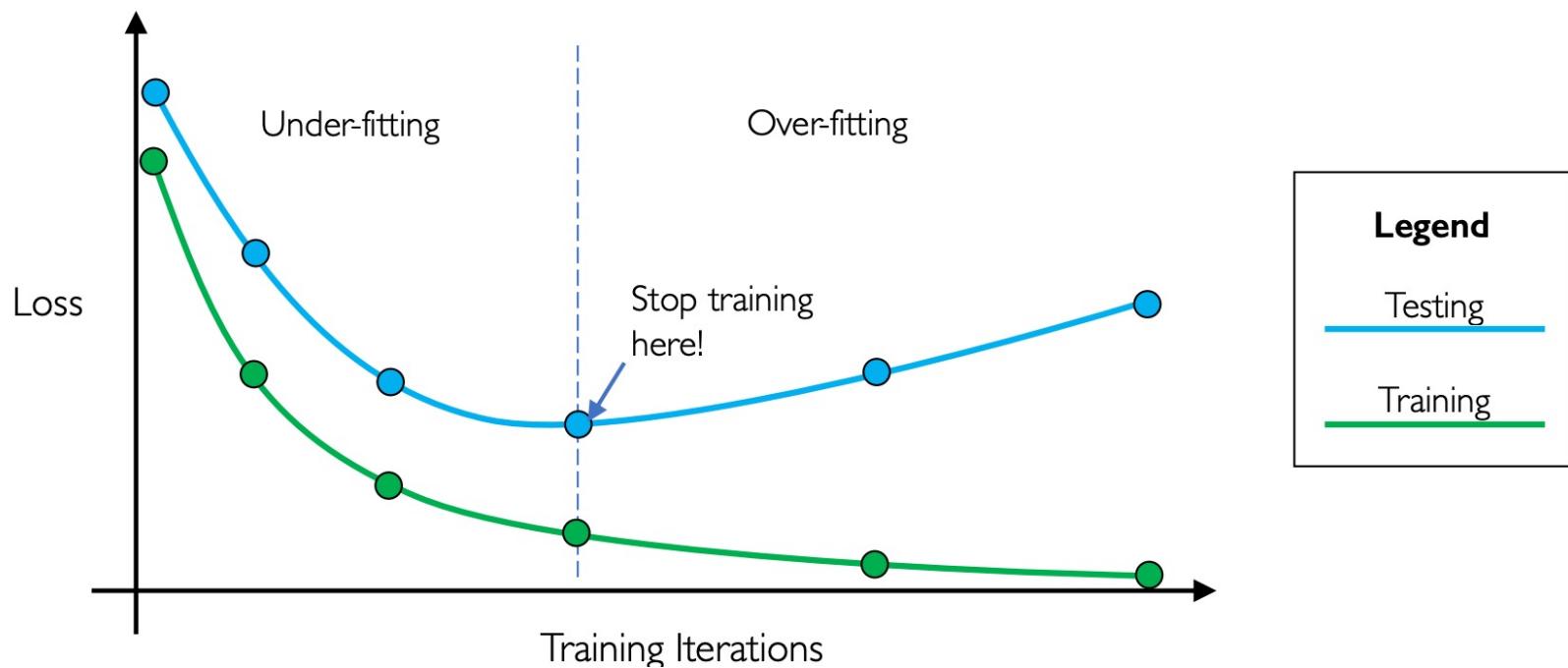
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

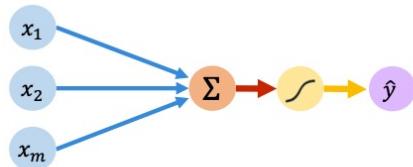
- Stop training before we have a chance to overfit



Core Foundation Review

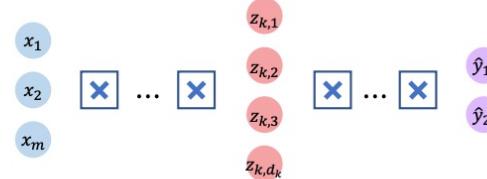
The Perceptron

- Structural building blocks
- Nonlinear activation functions



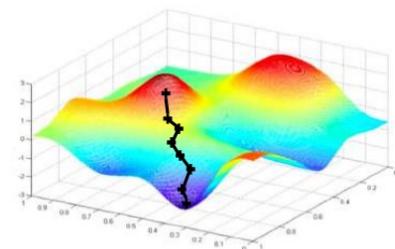
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Additional References

- Chapter 2 and Chapter 3 of [T2]
- Code example
 - <https://colab.research.google.com/drive/13cof4XUULbUqO0s5h-cd17FskWjpyMkm>